TURING 图灵原创

Docker 开发实践

曾金龙 肖新华 刘清 编著

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

曾金龙

计算机硕士，中山大学毕业，就职于迅雷网络，国内覆盖面最广的“迅雷P2P引擎”核心研发成员。研究方向为P2P网络、音视频传输和CEP系统。对Docker技术有着深入的理解，是国内较早将Docker引入到实际软件开发、测试和部署中的人。

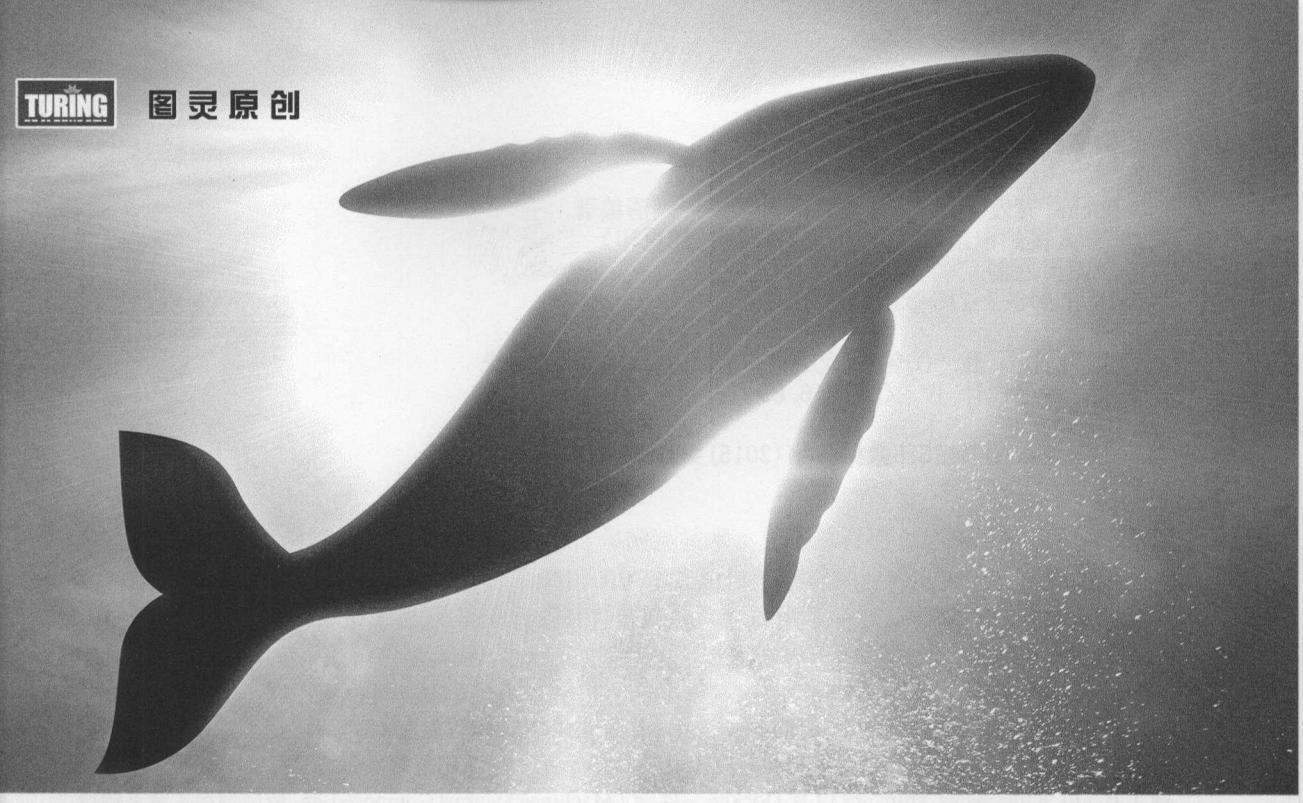
肖新华

工学学士，衡阳师范学院毕业，项目架构师。4年互联网软件开发经验，痴迷技术，对新生技术敏感。曾就职于迅雷网络、腾讯科技。

刘清

硕士，华中科技大学毕业，就职于迅雷网络，主要研究方向为移动下载库、音视频传输、调度策略设计等。

封面设计 | 83720326@qq.com
自在书装设计



Docker 开发实践

曾金龙 肖新华 刘清 编著

人民邮电出版社
北 京

图书在版编目（CIP）数据

Docker开发实践 / 曾金龙，肖新华，刘清编著. --
北京 : 人民邮电出版社，2015.7（2016.12重印）
（图灵原创）
ISBN 978-7-115-39519-1

Ⅰ. ①D… Ⅱ. ①曾… ②肖… ③刘… Ⅲ. ①Linux操作系统－程序设计 Ⅳ. ①TP316.89

中国版本图书馆CIP数据核字(2015)第122053号

内 容 提 要

本书由浅入深地介绍了 Docker 的实践之道，首先讲解 Docker 的概念、容器和镜像的相关操作、容器的数据管理等内容，接着通过不同类型的应用说明 Docker 的实际应用，然后介绍了网络、安全、API、管理工具 Fig、Kubernetes、shipyard 以及 Docker 三件套（Machine+Swarm+Compose）等，最后列举了常见镜像、Docker API 等内容。

本书适合 Docker 开发人员阅读。

◆ 编　　著　曾金龙　肖新华　刘　清
责任编辑　王军花
责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京九州迅驰传媒文化有限公司印刷
◆ 开本：800×1000　1/16
印张：18.25
字数：431千字　　2015年 7 月第 1 版
印数：4 601 - 4 900册　　2016年 12 月北京第 3 次印刷

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

序

在互联网这个领域，每年都会涌现出非常多的新技术，其中总有那么一两门是引领潮流的，例如前些年的 Hadoop 和如今的 Docker。热门的技术之所以能够得到业内的推崇，主要是因为它解决了行业痛点。Hadoop 解决的是分布式计算的问题，它提供了一套分布式存储和计算的解决方案，而且这个方案很廉价但很高效；Docker 解决的是服务器应用快速构建、部署和分享的问题，它能够把服务器应用像 APP 一样简单地安装到各种平台环境中，而不用受真实环境的影响。细说起来，它有 3 个核心概念：容器、镜像和 Docker Hub。容器提供一个隔离的安全运行环境，使得不同应用之间不会相互干扰。和容器经常对比的是传统的虚拟机技术，这二者侧重点不一样，但在资源节省方面，容器占极大优势。镜像是容器的静态存在形式，就好比程序是进程的静态文件形式一样。Docker 的镜像采用分层机制，每次改变就增加一层，用来记录这些改变，这样在传输的时候只需要传输改变的层即可。Docker Hub 是一个公共的镜像平台，为镜像分享提供便利。用户可以根据自己的需要，在已有的镜像上定制自己的镜像。除了这 3 大核心组件外，由于越来越多开发者加入到 Docker 的生态圈中，出现了越来越多的工具，例如 Fig、Kubernetes、shipyard 等。这些都是非常优秀的辅助工具，而 Fig 也被 Docker 官方收购，成为了官方的 Compose 服务，对外提供 Machine+Swarm+Compose 整套服务。

Docker 发展很快，网络上的资料也比较少，有的也只是一些初级操作，此时适时宜的一本好书对一门技术的布道也极为重要。而我看完本书之后，觉得它可以堪此大任。本书把内容分为了三篇：基础篇、案例篇和高级篇。基础篇向我们介绍了 Docker、容器、镜像等核心内容和操作，这部分的亮点在于体系性，对命令详细的解释以及对命令的体系编排，这是其他书所没有的。案例篇为我们提供了一些综合案例，每一个案例都是精挑细选的，所有的案例都非常具备代表性。在高级篇，本书的深度就体现出来了，它不是简单地给读者罗列几个概念和工具，而是在介绍完概念后进行实实在在的操作，这是目前任何关于 Docker 的书都没有的。所以，Docker 的高级使用者应该会对这部分内容非常感兴趣。例如，Docker API 编程、Fig、Kubernetes、shipyard 等，本书都给出了很完整的介绍和操作。总的来说，这本书在体系编排、广度和深度上都做得非常好。作者的文笔也很干练，没有过多的兜兜转转，这也是技术类图书该有的风格。这是一本关于 Docker 难得的宝典，无论你是初级用户还是高级用户，本书都能够带你走进 Docker 的世界，登堂入室。

甘南南

迅雷看看副总裁

2015 年 3 月

推 荐 语

好的技术需要一本好书去布道，让更多的开发者从中受益。本书通过基础、案例到高级话题三篇，带领着读者由浅至深，登堂入室。

——罗笑南，中山大学信息科学与技术学院教授

Docker是当之无愧的Go语言杀手级应用，并且现如今Docker这个词的含义越来越丰富了，以至于它已经代表了容器技术的生态圈。本书奉行实践出真知，其中的案例都非常棒。更为关键的是，其中的高级篇对 Docker生态圈的各个新成员也做了非常详实的介绍和实践，这真的很难能可贵。

——郝林，Go语言北京用户组发起人，《Go并发编程实战》作者

本人是在互联网行业摸爬滚打了十几年的老兵，我们开发的视频搜索服务，曾经占据了日本九成以上的市场，并被日本雅虎、乐天等主流公司采用。在此过程中，我接触到了日本大部分的主流互联网公司，深深地感觉到，应用的灵活部署、方便迁移、灵活扩充等是很大的难题，这方面的能力也是一个公司的核心竞争力。我们自己以及日本的合作伙伴，都为此做了许多的探索。

Docker是解决这些问题的很好的方案，是互联网行业的利器，今后必然会得到广泛的应用。

本书既有高屋建瓴的技术理念，也有实际操作的详尽说明，深入浅出了说明了 Docker 的相关知识，是一本不错的好书，值得每一位对Docker有兴趣的人收藏。

——颜文远，技术大牛

互联网技术每天都在快速更新，而Docker无疑是2014年最热门的互联网技术。这本书是我见过讲解Docker最为全面而又深入的图书，各个方面都有涉及，而且以实战为主。

——赵伟，北京精益智慧教育科技CTO

这是一本关于 Docker 的好书，值得所有想了解 Docker 的人放在键盘左边。

——李毅秋，人人网技术总监

腾讯的互娱的开发节奏，只有 Docker 跟得上！如果你想你的团队加快开发速度，那么我推荐你使用 Docker，而本书从基础、案例到高级话题，都有很全面的覆盖。

——易剑，腾讯互动娱乐事业群高级架构师

后台工程师（特别是运维工程师）都应该了解和熟悉 Docker 这一利器，使用它能极大地提高开发和部署效率。这本书通俗易懂，对于学习 Docker 的人非常有帮助！

——罗海江，大疆创新科技有限公司高级技术经理

云计算的初级是数据的云化，下一步是程序的云化，而 Docker 则是程序云化当前最好的工具。让你的程序一次配置，全网增量迁移、运行。本书出自一线互联网研发人员之手，它是实战的结晶，所涉案例都是互联网公司的真实应用，对 Docker 的应用都不是浅尝辄止，而是带你登堂入室。

——潘向荣，迅雷看看高级技术经理

前　言

Docker从提出到现在已经走过了两年的时间，在这两年里，它一直都是云计算领域的热点。可以说，它是2014年互联网最热门的技术。Docker得到Google、微软、IBM、Red Hat的声援，而它也不负众望，在这短短的两年里快速迭代，一步步变得更加完善。在Docker之前，开发者都深陷软件环境的配置之苦，虽然说并不是所有的软件配置都很难，但不同环境下的配置问题却层出不穷，相信很多读者和我有类似的经历，就是想用一款开源软件，结果配置了许久都跑不起来，然后不得不放弃用它。而网上总是有很多的答疑，可是跟着照做，发现在自己的机器上就是跑不起来。环境差异，可能会让原本简单的问题复杂化，迟滞我们的开发进程。拿来主义对于懒惰的程序员来说是件好事，我们都希望拿来就用，这样就可以专注于我们本该干的活。对于测试人员和运维人员来说，也是如此，没人喜欢处理这些本不该重点关注、处理不好却会让人寸步难行的问题。Docker就像一个打包器，可以把你的应用及其环境整体打包，然后很方便地迁移到不同的平台，到处运行。在用户看来就如同运行在原来的机器上一样。或许该有人说我用虚拟机也可以实现同样的效果，为什么要选择Docker。当两样东西都能够做同一件事情时，我们比的是效率。你可以在一台服务器上部署几个或者十几个虚拟机实例，但我相信没人会在一台服务器上部署上百个虚拟机实例，这是因为资源的限制。而在一台服务器上部署上百个Docker容器却并不是什么难事。在镜像的传输和共享方面，Docker也做得非常好，它能够只传输那些改变了的数据，而不用像传输虚拟机镜像那样，动辄至少几百兆。在共享方面，Docker建立了Docker Hub，你可以根据已有的镜像定制自己的镜像，而无需每次都再造轮子。如此接地气的技术，怪不得业内都惊呼Docker是下一个Hadoop。

本书的起源

虽然Docker人气旺盛，但关于Docker的书却少之又少，更别说汗牛充栋了，这主要是Docker出现的时间尚短。对于已有的书，基础内容不够体系，大多只停留寥寥几个基础命令的展示，并没有很好地归纳整理；而高级篇又只停留于粗浅的概念介绍，毫无实践价值，特别是对Docker具有很大作用的管理工具，例如Fig、Kubernetes、shipyard等内容，没有一本书去系统讲解它们。我们觉得这么好的技术，应该有更为系统的书去让更多的人了解它，理解它，这正是本书存在的价值。

如果不是遇到了王军花编辑，或许就没有本书，而正是在她的鼓励下，才让我们有勇气去将一些原本零碎的知识归纳整理为一本完整的书。

本书内容

本书主要介绍了 Docker 的实践之道。我们按照由浅入深的编排将本书分为三篇。在基础篇，主要是让读者认识 Docker 的概念和基础操作，对比介绍了 Docker 和虚拟机等技术，从容器、镜像、数据卷以及容器的连接等方面说明 Docker 的操作。通过对基础篇的学习，读者不仅对 Docker 有了全局的认识，而且能够对 Docker 的基础操作得心应手。该篇包含第 1 章至第 4 章的内容。

- 第 1 章从概念上介绍了 Docker，让读者对它的概念、背景、组件以及相关技术有了全局的认识。
- 第 2 章和第 3 章分别介绍了容器和镜像的相关操作，二者是 Docker 操作的核心对象。
- 第 4 章介绍了容器的网络基础、数据卷的配置以及多个容器之间的互联。

第二篇是案例篇。在这一篇中，我们通过不同类型的应用来说明 Docker 的实际应用。我们不做案例的简单堆砌，而是通过不同类型的案例来说明各个知识点的应用，它包含第 5 章至第 11 章。

- 第 5 章介绍了如何创建 SSH 服务镜像，这满足了日常 SSH 远程登录的需求。
- 第 6 章构建了一个采用 Apache 作为 Web 服务器、PHP 作为 Web 开发语言、MySQL 作为数据库的 Web 应用案例。
- 第 7 章构建了一个采用 Node.js 作为开发语言、MongoDB 作为数据库的 Web 案例，该案例着重说明了跨主机的多容器代理互联。
- 第 8 章和第 9 章说明了如何在公共云平台——阿里云上部署 Docker 应用，这里以 WordPress 为例进行介绍。
- 第 10 章介绍了如何使用私有仓库。
- 第 11 章将云计算的两大热点联合，说明了如何通过 Docker 来构建 Hadoop 镜像及集群。

第三篇是高级篇。在这一篇中，对 Docker 的 API 以及管理工具 Fig、Kubernetes、shipyard 以及 Docker 三件套（Machine+Swarm+Compose）都有实践操作，该篇包含第 12 章至第 18 章。

- 第 12 章介绍的是容器的高级网络知识。
- 第 13 章从命名空间、cgroups、Linux 能力机制以及服务端防护等方面入手介绍了安全方面的知识。
- 第 14 章则是通过 curl 工具来学习 Docker 的 API 接口，并给出 docker-py 库的编程实例。
- 在第 15 章至第 18 章中，我们分别介绍了 Fig、Kubernetes、shipyard 以及 Machine+Swarm+Compose 三件套，这些都是为了更好地管理和使用 Docker 的工具。

第四篇为附录。在附录 A 中，我们按照系统镜像、数据库镜像、Web 镜像、语言镜像的类别来编排，列举了常见的镜像，以供读者查阅。附录 B 是 Docker API 列表的归纳整理，分为容

器相关和镜像相关，亦是为了方便读者查阅。附录 C 是我们在写作过程中所用到的资料引用。

阅读须知

阅读本书时，最好能够从前往后依序阅读。特别是基础篇，是理解案例篇和高级篇的基石，读者最好能够读完基础篇，再阅读案例篇和高级篇。在案例篇和高级篇中，章和章之间的联系并没有那么紧密，例如第 6 章的案例和第 7 章的案例并没有关联，它们属于侧重点不同的案例，所以无需依序阅读。高级篇的工具也是如此，读者可以根据需求的迫切程度而选择性阅读。

目标读者

一切想了解及深入理解 Docker 技术的人，都是本书的目标读者。对于初级读者，通过本书的基础篇，你就可以成为一个能够灵活应用 Docker 的人。当你对 Docker 有一定了解后，通过学习案例篇和高级篇，你也就可以登堂入室了。

致谢

本书是很多人共同劳动的成果。三位作者要感谢一些人，感谢他们为本书所作出的巨大贡献和支持。

感谢在迅雷的同事：特别感谢甘南南和潘向荣两位的工作支持和悉心指导，感谢他们的慷慨大方，提供给我们充足的时间来写作本书。此外，还需感谢的同事有涂海涛、李明良、吴小强、易萌萌、何赞裕、吴建国和何锐，感谢他们在成书过程中的宝贵意见。

特别感谢本书的编辑王军花，没有她，本书就只能停留在笔者的脑海里。她不仅仅是一位资深的计算机类图书编辑，更像一名计算机专家，本书很多内容都是在她的指点下得以完善。

本书还需致谢的人有：盛建强博士、高怀恩博士、广发证券信息部的刘润佳和周英贵、美团网的蒋朋、百度的徐则水和罗剑波、腾讯的杨晓颖和郑克松、阿里巴巴的田晓娇以及李海龙、邱俊凌、杨文武、刘汇洋、冯学汉等人。感谢他们让各自公司相关平台成为本书部分案例的尝鲜者，并提出了诸多宝贵意见。

最后，感谢我们的家人。没有你们的支持，就没有这一切。

目　录

第一篇　基础篇：Docker基础

第二篇 案例篇：综合案例

第三篇 高级篇：高级话题、API、工具及集群管理

第四篇　附录

Part 1 第一篇

基础篇：Docker 基础

本篇内容

第1章 Docker简介

从虚拟机到容器，再到现在Docker的出现，虚拟化技术越来越受到互联网业界的关注和看好。在本章中，我们简要介绍一下Docker，其中主要包括以下内容。

Docker简介，其中包括Docker的概念、产生的背景、两个比Docker更早的虚拟化技术（虚拟机和容器）及其异同、Docker相对于一般容器的改进和优点以及Docker的应用场景。

- Docker的架构和组件。Docker是一种C/S架构的容器引擎，包含镜像、容器和库这3个重要概念。
- Docker的相关技术，主要从隔离性、可度量性、移植性和安全性这4个方面讨论。
- Docker的安装，其中包含各种Linux变种系统上的安装以及通过虚拟工具在Windows和OS X上的安装。

1.1 Docker 简介

在互联网初期，几乎所有的应用都以协议栈堆叠的形式进行开发，并且部署到单一的专有服务器上。如图1-1所示，15年前的应用又笨又重，而当时的终端设备也非常笨重，应用是基于一系列良好定义的协议栈进行开发的，它们包含中间件、运行时环境和操作系统；硬件所在的硬件环境也完全一致，即为一个服务配置单一的专有服务器，也就是说脱离了它原本的环境，整套系统或许就不能正常工作。随着互联网的发展，这种模式越来越不能满足日益复杂的互联网环境和产品需求。今天，应用开发者可以通过组合不同的服务来构建和装配应用，并使得应用能够跨越不同的硬件环境，如公共的、私有的以及虚拟的云服务器。

做到既能够组合当前最佳服务又跨越多种运行环境并非容易的事情。图1-2展示了当前一个网络应用可能涉及的方方面面，在软件层面，它的前台可能采用Nginx 1.5+ModSecurity+OpenSSL+Bootstrap 2来构建，后台采用Python 3.0等来构建，而在API端可能采用Python 2.7，数据库方面可能有多种数据库存在，每一项都是拿现存已有的服务，进而装配出应用。在硬件层面，它面对的环境错综复杂，可能是在虚拟机上部署，也可能在公共云、开发者的个人电脑、测试服务器以及产品集群等上部署。当一个应用拥有复杂的软件依赖关系和多样的硬件运行环境时，有以下几个问题必须面对。

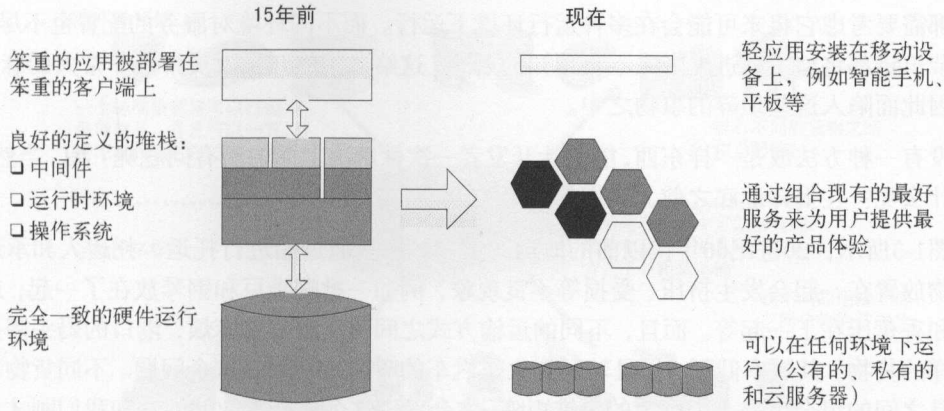


图1-1 互联网应用的演变

- ❑ 是否能够处理应用依赖的多样性和依赖库之间的不良反应？
- ❑ 是否能够适应硬件环境的多样性？
- ❑ 服务和应用之间的交互是否合理？
- ❑ 是否可以在多个平台之间快捷移动？

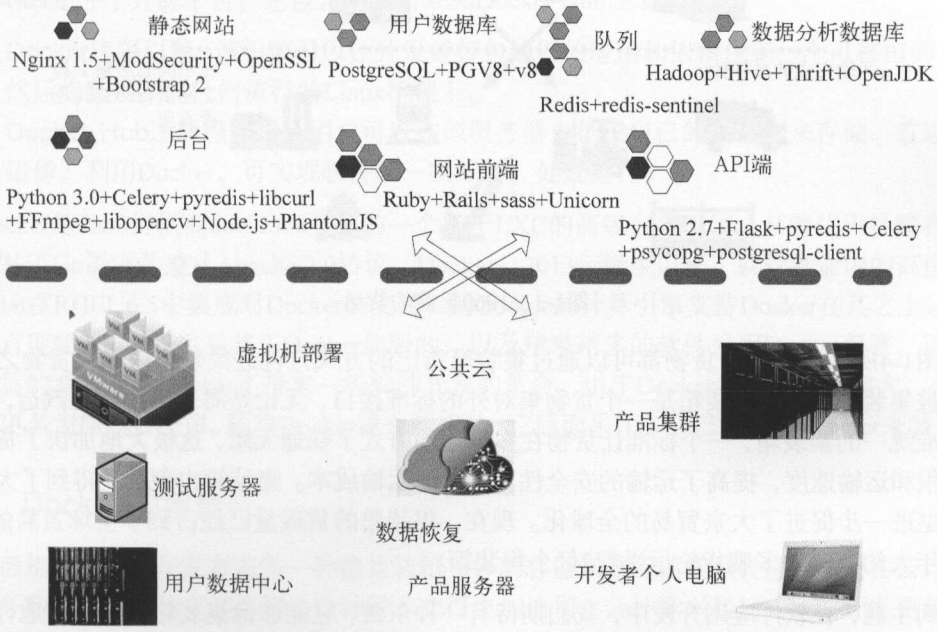


图1-2 应用的软件依赖和硬件运行环境的复杂性

想象一下将图1-2所示的各项应用和服务部署到各种硬件环境中的组合，每一个服务都有可能被部署在硬件环境中的一种甚至是多种。这些服务和运行环境的组合是一个可怕的矩阵，每一

个服务都需要考虑它将来可能会在多种运行环境下运行。而不同环境对服务的配置也不尽相同，一个环节出错都将给工程进度带来不可预知的迟滞。这给应用的开发人员带来非常大的麻烦，他们可能因此而陷入这些琐碎的事物之中。

有没有一种方法或是一样东西，能够让开发者一次性解决上面的所有问题呢？有，当然存在！那它是什么呢？在揭开谜底之前，我们先卖个关子。

如图1-3所示，20世纪60年代以前的海运，大多数散货通过船进行托运，托运人和承运商都担心货物放置在一起会发生挤压、受损等不良现象，例如一批咖啡豆和钢琴放在了一起，或是一批钢材和香蕉压在了一起等。而且，不同的运输方式之间转运也非常麻烦，港口的码头需要装卸各式各样的货物，其效率低下，而且转运到火车汽车的时候也得面临这个问题。不同货物和不同交通工具之间的组合也是一个巨大的二维矩阵，你会发现这个传统行业的问题和我们刚才遇到的问题是如此的一样！海运界最后在美国海陆运输公司的推动下，制定了国际标准集装箱来解决这个棘手的问题。

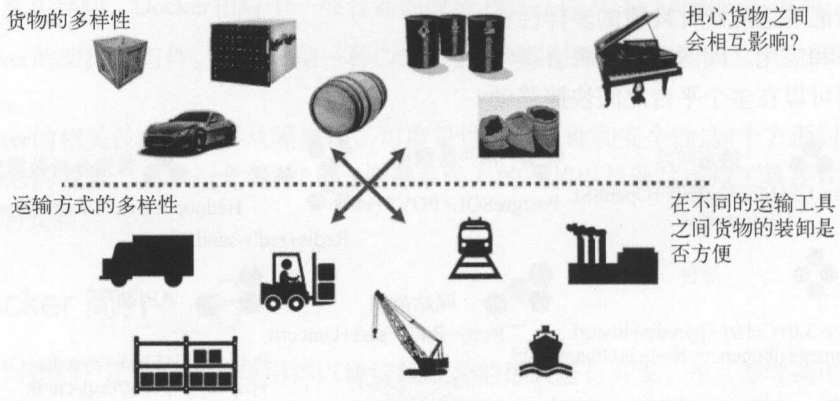


图1-3　1960年前的货运

如图1-4所示，所有的货物都可以通过集装箱指定的方式打包进集装箱内部，货物之间的相互影响被集装箱隔绝。集装箱是一个货物集对外的标准接口，无论是海运码头还是汽运，处理的都是标准统一的集装箱，一个标准让货物在多种运输方式下畅通无阻，这极大地加快了货物的装卸、堆积和运输速度，提高了运输的安全性，降低了运输成本。集装箱出现后，得到了大规模的推广，也进一步促进了大宗贸易的全球化。现在，集装箱的货运量已经占到了全球贸易的90%以上，每年大约有5000多艘货轮运送着2亿个集装箱。

回到主题，在软件应用开发中，我们期待有一种东西，它能够像集装箱一样方便地打包应用程序，隔离它们之间的不良影响，使应用能够在各种运行环境下运行并且在平台之间易于移植。Docker，正是这个集装箱。在Docker出现之前，类似的技术已经存在，例如虚拟机和容器，然而它们能够解决前面4个问题里面的其中某些，却不能解决所有，这也正是它与众不同和受到热捧的重要原因。接下来，我们进入Docker的主题。

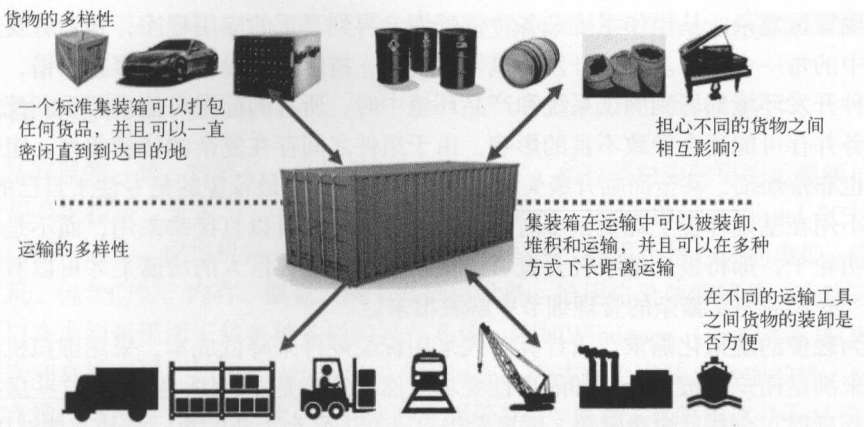


图1-4 标准集装箱的出现解决了运输方面的难题

1.1.1 Docker 的概念

什么是Docker?

Docker是一个开源平台，它包含容器引擎和Docker Hub注册服务器。

- **Docker容器引擎**：该引擎可以让开发者打包他们的应用和依赖包到一个可移植的容器中，然后将其发布到任何流行的Linux机器上。
- **Docker Hub注册服务器**：用户可以在该服务器上创建自己的镜像库来存储、管理和分享镜像。利用Docker，可实现软件的一次配置、处处运行。

Docker是PaaS提供商dotCloud开源的一个基于LXC的高级容器引擎，其源代码托管在GitHub上，它基于Go语言并遵从Apache 2.0协议。Docker自2013年诞生以来，就受到业内的高度关注，从RedHat在RHEL 6.5中集成对Docker的支持，到Google的计算引擎支持Docker在其之上运行，再到国内百度的App引擎也是基于Docker部署的，以及越来越多的软件采用Docker部署。可以说，在云计算的背景下，Docker正带来一场软件开发的革命。由于Docker的影响越来越大，dotCloud公司也更名为Docker公司。随着Docker越来越成熟，它后面采用了自己的libContainer来替换LXC。

1.1.2 Docker 的背景

前面提到，软件开发者急需一种像集装箱一样的容器来装配运输软件应用。而在云计算兴起后，服务和运行平台越来越多样，这种需求变得更加迫切。云计算兴起后，软件开发更趋向于组件组装，选取最合适的服务集合装配到一起构建出最终的产品，并将应用部署到各类云平台之上。云计算平台对硬件进行抽象和虚拟，按量提供给开发者，使得开发者从硬件管理问题中解脱出来，典型的云计算平台有亚马逊的AWS、国内的阿里云等。在云计算时代，硬件的部署和管理问题已经得到了很好的解决，然而软件的依赖、部署和管理问题依然存在，这主要体现在以下几个方面。

- **环境管理复杂。**从操作系统到各种依赖库，再到真正的应用程序，开发者要关心整个环境中的每一个环节。有些开发环境特别复杂，搭建过程漫长而且容易出错，而当需要将这种开发环境部署到测试系统和产品环境中时，所有的配置可能需要重新配置。不同的服务并存可能还会导致不良的影响，由于组件之间存在复杂的依赖关系，组件的版本升级也非常烦琐，甚至面临升级失败。事实上，开发者最希望的是专注于自己的业务逻辑，而不用在基础环境上耗费太多时间。如果基础环境可以直接拿来用，而不是每次都重新发明轮子，那将极大地减轻开发者的负担，毕竟站在巨人的肩膀上才可以看得更远。因此，我们需要从繁杂的管理细节中解脱出来。
- **更为轻便的虚拟化需求。**云计算时代采用标配硬件来降低成本，采用虚拟机的虚拟化手段来满足用户的按量分配和隔离性要求。然而无论是KVM还是Xen，这些虚拟化软件技术都显得冗余并且浪费资源。用户希望自己在云平台上花的每一分钱都能够用在刀刃上，而不是将大部分钱耗费在一个个的操作系统和一些公共库上面。所以，在云计算平台上，用户希望出现一个比虚拟机更为轻便的虚拟手段。
- **移植性的需求。**容器是一种比虚拟机更为轻便的虚拟技术，例如LXC。Docker本身也属于容器的一种。容器可以解决资源浪费问题，然而在Docker出现之前的容器，并不是为云计算环境设计的，在可移植性方面做得不好，而且难于配置。用户希望自己的容器可以部署到尽可能多的平台上去，而又不用关心平台差异。显然，传统的容器都不能胜任。

新的需求驱动新的技术，Docker为问题而生，作为一个变革者，做了这个众人期待的“集装箱”。如图1-5所示，Docker把运行环境比作海运，把操作系统比作货船，而把Docker容器比作集装箱。Docker容器也就是软件的集装箱，Docker以一套标准的方法让用户可以构建自己的容器，然后容器易于在各大平台移植和运行，Docker对应用也提供了良好的隔离性。Docker容器采用组件的方式构建，用户可以像堆积木一样，基于现有的容器来增加自己的应用，从而构建出自己的容器，Docker的增量式更改减少了容器的冗余，使得Docker容器易于修改和传输。

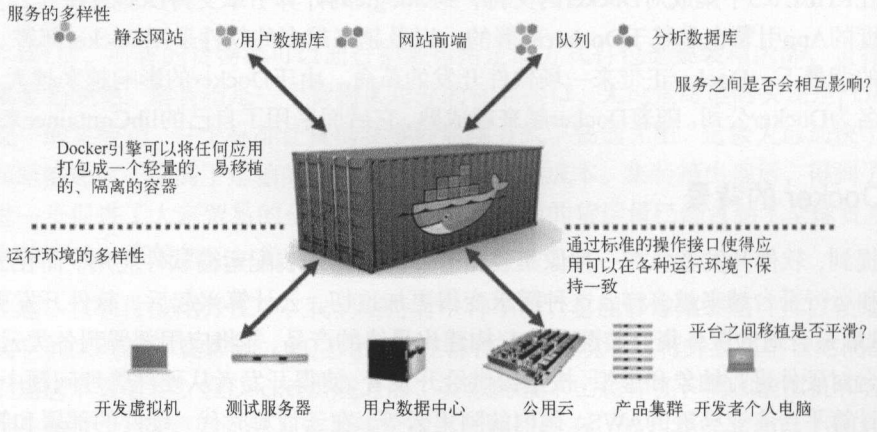
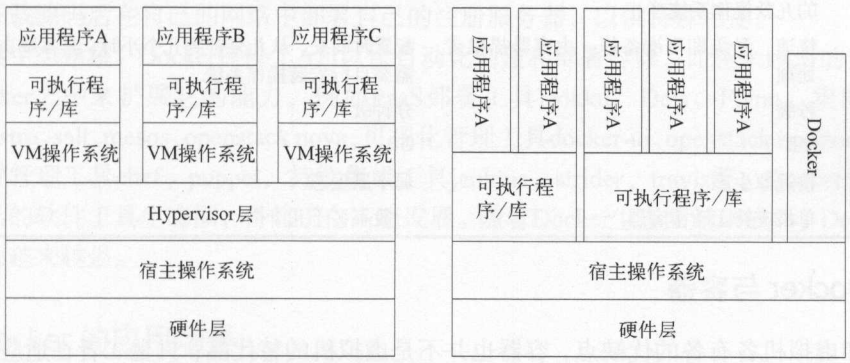


图1-5　Docker是一个软件应用的集装箱

Docker是如何做了这个变革者的呢？下面我们通过容器和虚拟机的对比说明一下。

1.1.3 容器与虚拟机

Docker是容器的一种，容器是一种轻量级的虚拟技术，和容器对应的更为重量级的虚拟技术是虚拟机。提及虚拟机，大家肯定都不陌生，例如VMware、VirtualBox、Virtual PC这些耳熟能详的产品早已深入人心。虚拟机是一种基于硬件的虚拟技术，它采用指令级的虚拟，完全虚拟一整套物理主机，包含CPU、内存、磁盘、网卡等硬件设备，给用户呈现的就是一个物理主机的特性。用户可以在虚拟机里面安装各种各样的操作系统，例如Windows、Linux或者是OS X，一切操作都看起来和真机一样。如图1-6a所示，用户可以在一台主机上安装多个虚拟机，每一个实例都完整地包含硬件虚拟层、操作系统、公共库和应用等部件。然而，有时候这并不是我们想要的，因为这样太耗费资源，而且管理也并不方便。还是那句话，当新的需求出现后，自然会驱动新的技术出现，于是后来出现了一种更为轻量级的虚拟技术——容器。容器是一种基于操作系统的虚拟技术，它运行在操作系统之上的用户空间，所有的容器都共用一个系统内核，甚至是公共库，容器引擎提供进程级别的隔离，让每个容器都像运行在单独的系统之上，但又能够共享很多底层资源，如图1-6b所示。比起虚拟机，容器更为轻量、快速、易于管理。除了Docker，常见的容器还有Solaris Zones、BSD jails、OpenVZ和LXC等。



(a) 虚拟机　　(b) Docker容器

图1-6 虚拟机和Docker容器的对比

为了更详细地对比虚拟机和容器，我们把二者的异同点列在表1-1中。

表1-1 容器与虚拟机的对比

	容　器	虚拟机
相同点	❑ 都可在不同的主机之间迁移 ❑ 都具备root权限 ❑ 都可以远程控制 ❑ 都有备份、回滚操作	

（续）

	容　器	虚拟机
操作系统	在性能上有优势，能够轻易地同时运行多个操作系统	可以安装任何系统，性能不及容器
原理	和宿主主机共享内核，所有容器运行在容器引擎之上，容器并不是一个完整的操作系统，所有的容器共享操作系统，在进程级进行隔离	每一个虚拟机都建立在虚拟的硬件之上，提供指令级的虚拟，具备一个完整的操作系统
优点	更为高效、集中。一个硬件节点可以运行数以百计的容器，非常节省资源。QoS会尽量满足，但不保证一定满足。内核由提供者升级，服务由服务提供者管理	对操作系统具有绝对的权限，对系统版本和升级具有完全管理权限。具有一整套的资源：CPU、RAM和磁盘。QoS是有保证的。每一个虚拟机像一个真实的物理机一样，可以实现不同的操作系统同时运行在同一物理节点上
资源管理	弹性的资源分配：资源可以在没有关闭容器的情况下添加，数据卷也无需重新分配大小（有些服务的容器需要重启）	虚拟机需要重启，虚拟机里面的操作系统需要处理新加入的资源。例如，添加一块磁盘，则需要重新分区等
远程管理	根据操作系统的不同，会通过shell或者远程桌面进行。前提是容器内的操作系统已经启动	远程控制由虚拟化平台提供，可以在虚拟机启动之前连接。所以可以安装系统
缺点	对内核没有控制权限，只有容器的提供者具备升级权限。只有一个内核运行在物理节点上，几乎不能实现不同的操作系统混合。容器提供者一般仅提供少数的几款操作系统	每一台虚拟机具有更大的负载，耗费更多的资源，用户需要全权维护和管理。一台物理机上能够运行的虚拟机也非常有限
配置	快速，秒级即可准备好。由容器提供者处理	配置时间长，从几分钟到几个小时，具体取决于操作系统。需要自行安装操作系统
启动时间	秒级	分钟级
硬盘使用	MB	GB
性能	接近原生态	弱于原生态
系统支持量	单机支持上千个容器	一般不多于几十个

1.1.4　Docker与容器

容器和虚拟机各有各的优缺点，容器也并不是虚拟机的替代品，只是二者在适应不同的需求下各有优点。容器相对于虚拟机的优势在于效率更高，资源占用更小，管理更为便捷。在需要部署的系统都是同一系列的操作系统时，这种性能和便捷性上的优势非常明显。然而就容器本身而言，依然有很多问题需要解决，例如在灵活性、安全性以及配置共享方面都存在不足。我们以Docker早期基于的LXC容器为例，说明Docker对容器进行的改进和优化，其中很多方面是LXC这类容器所没有的新特性。

- **跨平台的可移植性**。Docker定义了一种统一标准的打包格式，将应用及其依赖打包进单个的镜像中，该镜像可以在任何Docker可运行的机器之间便捷传输，并且在不同机器上，Docker隔离了应用和平台的直接联系，对配置进行了抽象，使得在任何平台上应用的运行环境都一样。LXC的配置却不是可移植的，它依赖于某台具体的机器，例如机器的网

络、存储、日志、磁盘等配置，同一配置在不同机器上并不能够通用，这极大地限制了它的可移植性。

- **面向应用**。Docker是为优化应用部署而诞生的，LXC则是面向机器的，这是二者设计哲学上的不同。Docker面向应用的哲学体现在API、用户接口以及文档等各个方面，Docker都是为了更简单地做事。而LXC则更关注于使用更少的CPU、更少的RAM，成为一个更轻量的机器。
- **版本控制**。Docker的版本控制和git工具非常类似，Docker可以跟踪一个容器的版本信息，查看版本差异，提交和回滚版本等。所有的版本信息都将被记录，这样你可以清晰地看到一个应用服务器的更改历史。
- **组件复用**。Docker容器以组件式搭建，你可以利用一个基础镜像构建更多的应用容器。例如，你可以准备好一个Python的运行环境作为基础镜像，然后以该镜像为基础，构建不同的Web应用。你可以配置一个postgresql基础镜像，作为未来的数据库基础组件。这些配置可以手动处理，也可以自动化完成。
- **共享性**。Docker拥有一个公共的注册服务器。成千上万的开发者上传了他们的镜像，这些镜像涵盖软件的各个应用领域，我们可以通过这些共享的镜像来进一步定制自己的镜像。在这些开发者中，有一大批是官方组织提供的标准库，这些镜像安全、可靠，有持续的维护和版本升级。而Docker的注册服务器本身也是一个开源项目，所以任何人都可以下载源码后在自己的网络中部署自己的注册服务器，以供特殊用途。
- **工具生态系统**。Docker提供了API以供自动化创建和部署容器，而越来越多的工具加入到Docker之中来扩展它的能力，例如PaaS部署工具Dokku、Deis、Flynn，集群管理工具maestro、salt、mesos、openstack nova，可视化管理工具docker-ui、openstack horizon、shipyard，配置管理工具chef、puppet，持续集成工具jenkins、strider、travis等。Docker已经形成了自己的软件工具生态圈，目前正在高速发展。随着Docker生态圈越来成熟，Docker的能力将会越来越强。

1.1.5 Docker 的应用场景

Docker作为一款容器，并且比传统的容器具有那么多的改进和优点，那么我们可以拿它来做些什么？以下是几种典型的应用场景。

- 加速本地开发。通过Docker能够快速搭建好开发和运行环境，并且该环境可以直接传递给测试和产品部署。
- 自动打包和部署应用。
- 创建轻量、私有的PaaS环境。
- 自动化测试和持续集成/部署。
- 部署并扩展Web应用、数据库和后端服务器。
- 创建安全沙盒。

- 轻量级的桌面虚拟化。

1.2　Docker 的组件

Docker采用的是C/S架构，具体如图1-7所示。Docker客户端，即Docker可执行程序，可以通过命令行和API的形式与Docker守候程序进行通信，Docker守候程序提供Docker服务。

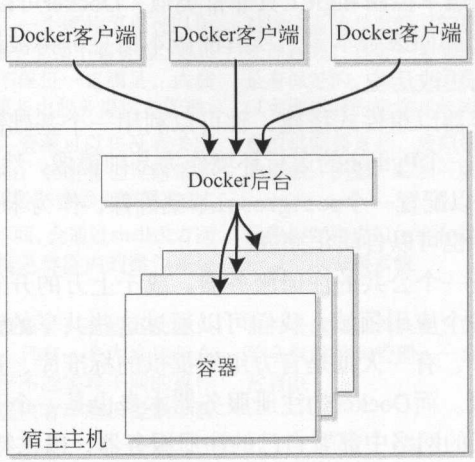


图1-7　Docker的C/S架构

Docker包含三大核心组件——镜像、容器和库。

- **镜像**：是一个只读的静态模板。它保存着容器需要的环境和应用的执行代码，可以把镜像看成容器的代码，当代码运行起来后就成了容器。镜像采用分层机制，每个镜像都是只读的，但是可以将写数据的层通过联合文件系统附加在原有的镜像之上。这种增量式修改使得镜像非常容易存储、传输和更新。本书将会涉及镜像的获取和制作。
- **容器**：是一个运行时环境，它是一个镜像的运行状态，相对于静态的镜像而言。容器是镜像执行的动态表现。用户可以在容器中运行所想要的程序和服务，而容器就像一个集装箱，它并不关心你运行的到底是什么程序，所有应用的运行方式都一样——创建、开始、停止、重启和销毁；容器也不在乎你在什么样的环境中运行它，可以在个人电脑、虚拟机、云服务器、各种操作系统上运行。容器易于交互、便于传输、易移植、易扩展，非常适合进行软件开发、软件测试以及软件产品的部署。
- **库**：Docker采用注册服务器来存储和共享用户的镜像，库是某个特定用户存储镜像的目录。通常，一个用户可以建立多个库来保存自己的镜像。从这里可以看出库是注册服务器的一部分，一个个的库组成了一个注册服务器。注册服务器有公共的和私有的，其中公共的如Docker官方的Docker Hub。注册一个账号，你就可以在里面建立自己的镜像库。镜像库可以选择开放，也可以选择私有，仅有被允许的组成员才可以访问。

图1-8是一个通过Docker进行应用开发和部署的案例流程，通过这张图可以很好地理解这3个概念。首先，在开发主机上构建容器A，构建方法既可以是手工构建，也可以通过Dockerfile自动构建。容器A的构建必须基于一个已有的基础镜像，并在它之上执行一系列操作。镜像A是容器的静态形式，容器是镜像的运行态。将容器A保存为镜像A，然后推送到Docker库中进行共享。这时候，在产品部署方面，另一端则可以通过在Docker库中搜索来获得镜像A，并将其拉取到本地，最后在产品集群中运行容器A，其中产品集群中一般会同时运行很多容器，例如容器B和容器C等，而这些容器互不影响，相互隔离。

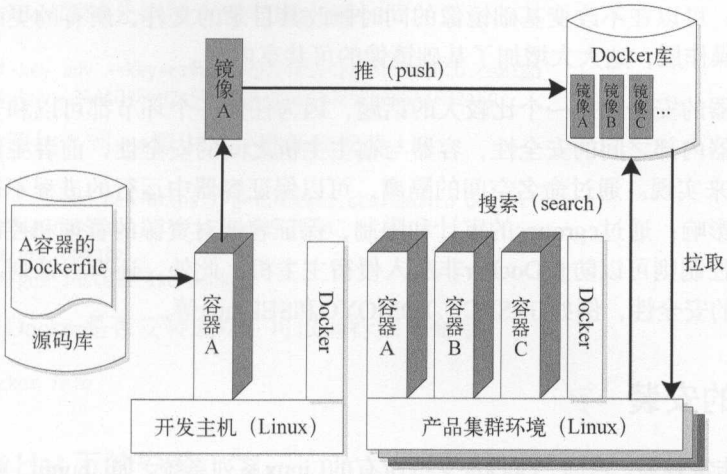


图1-8 使用Docker进行开发和部署的流程

1.3 Docker的相关技术

Docker是利用容器来实现的一种轻量级的虚拟技术，从而在保证隔离性的同时达到节省资源的目的。Docker的可移植性也让其能够无缝地运行在各种平台上，方便改写和传输。理解Docker这种虚拟技术，可以从其隔离性、可度量性、移动性和安全性这4个方面来探讨。

- **隔离性**：Docker采用libcontainer作为默认容器，取代了之前的LXC。libcontainer的隔离性主要是内核的命名空间来实现的，具体有pid、net、ipc、mnt和uts等命名空间，它们将容器的进程、网络、消息、文件系统和主机名进行隔离。
- **可度量性**：在虚拟机中，由于虚拟好了CPU、内存等硬件要素，每个用户根据自己的需求选择硬件的配置，所用的资源可以量化计算。在Docker中，主要通过cgroups（控制组）来控制资源的度量和分配。
- **移植性**：在虚拟机中，为了更好地复制、重建和移动虚拟机，采用了镜像和快照的方法。而像LXC这类容器，其迁移性非常差，每部署一台计算机，都可能需要重新配置，所以不能实现快速的大规模部署和更新。Docker利用AUFS来实现对容器的快速更新。AUFS

是一种支持将不同目录挂载到同一个虚拟文件系统下的文件系统，支持对每个目录的读写权限管理。另外，AUFS具有层的概念，每一次修改都是在已有的只写层进行增量修改，修改内容将形成新的文件层，而不影响原有的层。采用AUFS作为Docker容器的文件系统，能够提供如下好处。

- **节省存储空间**：多个容器可以共享同一个基础镜像存储。
- **快速部署**：当要部署多个来自同一个基础镜像的容器时，避免了多次复制操作。
- **升级方便**：升级一个基础镜像即可影像到所有基于它的容器。
- **增量修改**：可以在不改变基础镜像的同时修改其目录的文件，所有的更改都发生在最上层的写操作层，这大大增加了基础镜像的可共享内容。

- **安全性**：容器的安全性是一个比较大的话题，因为任何一个环节都可以和安全相关。这可以分为容器内部之间的安全性，容器与宿主主机之间的安全性，前者主要通过命名空间和cgroups来实现。通过命名空间的隔离，可以保证容器中运行的进程不被主机的进程和其他容器影响；通过cgroups的审计和限制，保证容器对资源的管理和控制安全。内核能力机制的控制则可以防止Docker非法入侵宿主主机。此外，还有一系列工具的组合来保证Docker的安全性，例如GRSEC、TOMOYO和SELinux等。

1.4　Docker的安装

Docker的安装非常容易。目前，Docker支持所有的Linux系列系统，如Ubuntu、RHEL、Debian等。通过Boot2Docker虚拟工具，在OS X和Windows下也能够运行Docker。

但目前而言，Docker的运行环境也有限制，具体如下。

- 必须是在64位机器上运行，并且目前仅支持x86_64和AMD64，32位系统暂时不支持。
- 系统的Linux内核必须是3.8或者更新的，内核支持Device Mapper、AUFS、VFS、btrfs等存储格式。
- 内核必须支持cgroups和命名空间。

接下来，我们说明各个操作系统平台下Docker环境的安装。需要说明的是，本书从第2章开始，如不特别说明，默认都是在Ubuntu Trusty 14.04(LTS)（64位）系统下进行的操作。

1.4.1　Ubuntu下的安装

Docker支持以下版本的Ubuntu系统：Ubuntu Trusty 14.04、Ubuntu Precise 12.04、Ubuntu Raring 13.04 和 Saucy 13.10。这里以Ubuntu Trusty 14.04(LTS)（64位）为例进行介绍。

Ubuntu Trusty的内核是3.13.0，在这个系统下安装时默认的Docker安装包是0.9.1。

首先，运行以下命令进行安装：

```
$ sudo apt-get update
$ sudo apt-get install docker.io
```

然后重启伪终端即可生效。

如果想安装最新的Docker，首先你需要确认你的apt是否支持https，如果不支持，则需通过如下命令进行安装：

```
$sudo apt-get update
$sudo apt-get install apt-transport-https
```

然后将Docker库的公钥加入到本地apt中：

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
   --recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E9
```

再将安装源加入到apt源中，并更新和安装：

```
$ sudo sh -c "echo deb https://get.docker.com/ubuntu docker main\
   > /etc/apt/sources.list.d/docker.list"
$ sudo apt-get update
$ sudo apt-get install lxc-docker
```

为了验证Docker是否安装成功，可以运行如下命令：

```
$ sudo docker info
```

1.4.2 Red Hat下的安装

红帽系列主要有红帽企业版 Linux 6、红帽企业版 Linux 7和Fedora。

1. 红帽企业版Linux 7（RHEL7）

红帽企业版Linux 7（RHEL7）中已经默认加入了Docker，它位于附加频道中。安装Docker时，首先要启用附加频道，相关命令如下：

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
```

然后进行安装，相关命令如下：

```
$ sudo yum install docker
```

2. 红帽企业版Linux 6（RHEL6）

首先，你要保证自己的红帽是6.5以上的，内核版本在2.6.32-431以上。

在红帽企业版Linux 6和CentOS 6中，我们首先需要安装EPEL包库，而在Fedora上却不用。对于不同的平台，一些包的命名和版本也不尽相同。

安装EPEL的命令：

```
$ sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386 ↵
```

```
/epel-release-6-8.noarch.rpm
```

然后安装Docker：

```
$ sudo yum -y install docker-io
```

3. Fedora上的安装

在Fedora 19上安装Docker的命令如下：

```
$ sudo yum -y install docker-io
```

在Fedora 20上安装Docker的命令如下：

```
$ sudo yum -y install docker
```

4. 在红帽系列中启动Docker后台

安装好之后，我们就可以启动Docker的后台服务了。

在RHEL 6和CentOS 6中，可以通过如下命令来启动：

```
$ sudo service docker start
```

我们还可以让Docker服务开机启动：

```
$ sudo service docker enable
```

在RHEL 7和Fedora系统中，则是：

```
$ sudo systemctl start docker
```

开机启动是：

```
$ sudo systemctl enable docker
```

我们可以通过如下命令来确认Docker是否已经安装成功：

```
$ sudo docker info
Containers: 13
Images: 22
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Dirs: 48
Execution Driver: native-0.2
Kernel Version: 3.13.0-24-generic
Operating System: Ubuntu 14.04 LTS
WARNING: No swap limit support
```

1.4.3　OS X下的安装

因为Docker引擎采用的是Linux的内核和内核特性，如果需要在OS X上运行它，则需要一个

虚拟机。Docker已经给我们提供了一种简洁方法，那就是利用Boot2Docker工具来安装虚拟机和配置Docker服务。这里的虚拟机指的是VirtualBox。

首先，可以去GitHub上下载最新的Boot2Docker。在编写这本书的时候，其最新版本是v1.3.2。

双击下载完的安装包，将会自动安装，如图1-9所示。安装的内容包含VirtualBox虚拟机、Docker和Boot2Docker管理工具。

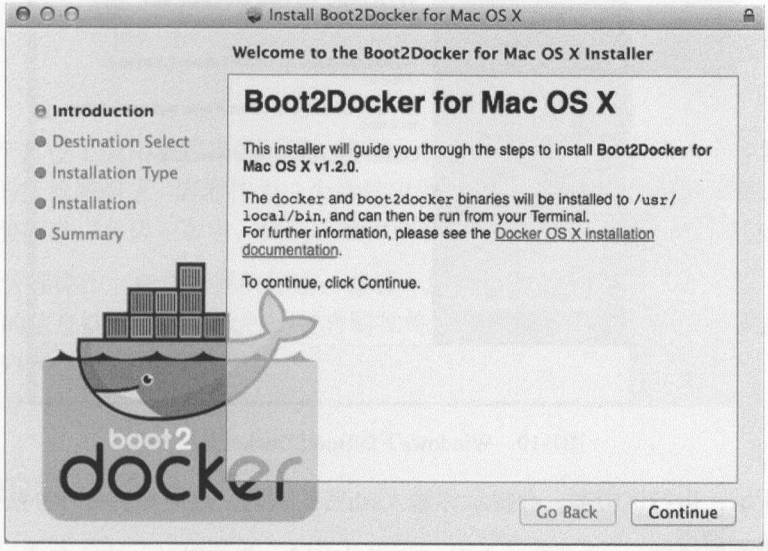


图1-9 OS X系统下Boot2Docker的安装

安装完成之后，你可以在OS X的“应用”文件夹中找到Boot2Docker。直接双击来启动它，或者采用如下命令行的方式：

```
$ boot2docker init
$ boot2docker start
$ $(boot2docker shellinit)
```

然后在终端可以验证Docker是否成功安装：

```
$ docker info
Containers: 13
Images: 22
Storage Driver: aufs
...
```

至此，OS X下的Docker就安装好了。

1.4.4 Windows下的安装

在Windows系统下运行Docker也需要虚拟机，我们也一样借助Windows版本的Boot2Docker

来安装。

首先，下载最新版本的Windows版的Boot2Docker，当前的最新版本为1.3.2，然后双击安装包安装即可，如图1-10所示。

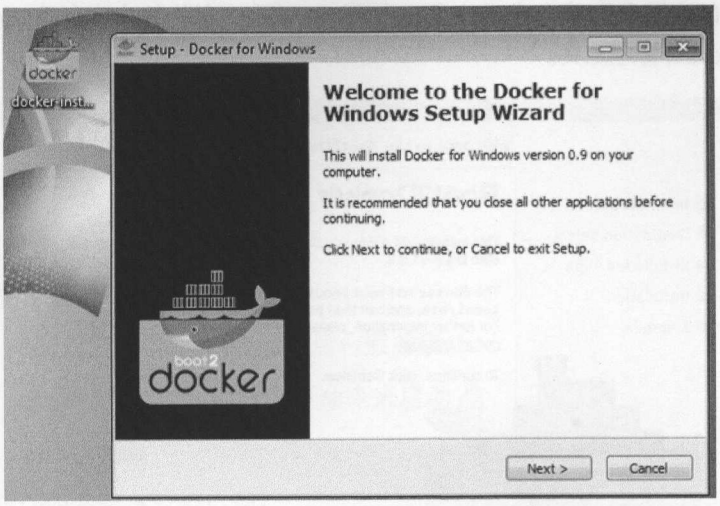


图1-10　Windows下的Boot2Docker安装

如图1-11所示，运行该脚本，会提示你输入ssh密钥密码，最简单的处理办法是暂时不管它，直接按回车[Enter]即可。

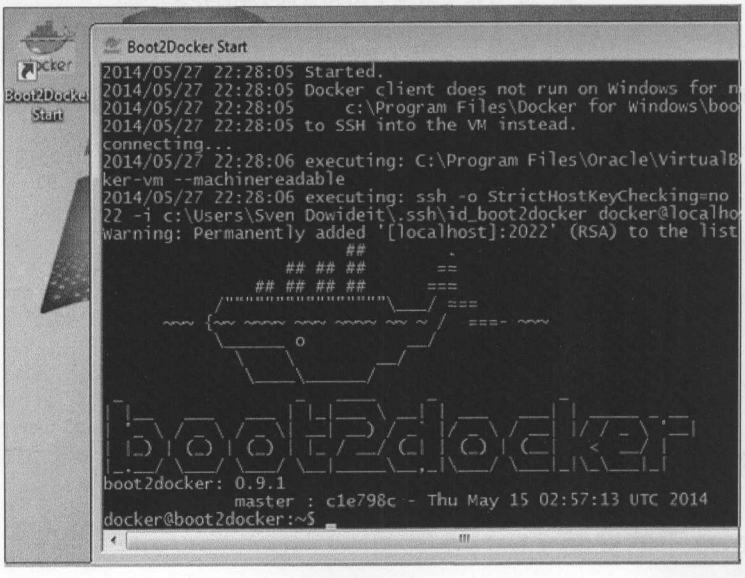


图1-11　Windows下的Docker运行

第2章 容 器

容器是一个打包了应用和服务的环境。它是一个轻量级的虚拟机，每一个容器都由一组特定的应用和必要的依赖库组成。容器作为一个软件应用的标准集装箱，它必然需要定义一组跟具体应用无关的标准接口。在这一章中，我们主要说明容器的常用标准操作，主要包含以下内容。

- 容器的管理操作。包含容器的创建、查看、启动、终止、删除。
- 容器内的信息获取和指令执行。包含附加终端到后台容器，查看容器日志和详细信息以及容器内执行指定的指令。
- 容器的导入和导出。

2.1 容器的管理操作

对于容器的常见命令（包括查看、创建、启动、终止和删除等），我们按照由浅到深、由必须到可选的顺序介绍。

要查看某条指令的详细帮助信息，可以访问http://docs.docker.com/reference/，或者通过docker help命令。此外，我们也可以通过man pages查看（例如：man docker-run）。

2.1.1 创建容器

创建容器有两个命令，一个是docker create，另一个是docker run。二者的区别在于前者创建的容器处于停止状态，而后者不仅创建了容器，而且启动了容器。

采用docker create创建一个停止状态的容器，具体如下：

```
$ docker create ubuntu:14.04
adedeb41185ad9ac096b8f1a7a3c2d3c2f0cf759643685d320ff656bd49b9d48
$ docker ps
CONTAINER ID    IMAGE          COMMAND    CREATED          STATUS       PORTS       NAMES
$ docker ps -a
CONTAINER ID    IMAGE          COMMAND    CREATED          STATUS       PORTS       NAMES
adedeb41185a  ubuntu:14.04  "/bin/bash"  27 seconds ago                             stupefied_brown
```

创建容器后，Docker会立刻返回容器的ID，例如adedeb...就是我们刚刚所创建容器的ID。ID

可以唯一标识一个容器，每一个容器的ID都是独一无二的。docker ps命令用于查看正在运行的容器，我们没有看到任何运行的容器，docker ps -a则是查看所有容器，包含未启动的容器。可以看到，我们创建的容器已经存在。

想要让创建的容器立马进入运行态，可以使用docker run命令，该命令等同于用docker create创建容器后再使用docker start启动容器。使用docker run命令，可以创建两种类型的容器——后台型容器和交互型容器。

- **交互型容器**：运行在前台，通常会指定有交互的控制台，可以给容器输入，也可以得到容器的输出。创建该容器的终端被关闭，在容器内部使用exit命令或者调用了docker stop、docker kill命令后，容器会变成停止状态。
- **后台型容器**：运行在后台，创建启动之后就与终端无关。即便终端关闭了，该后台容器也依然存在，只有调用docker stop或docker kill命令时才能够使容器变成停止状态。

下面我们创建一个交互型容器，相关代码如下：

```
$ sudo docker run -i -t --name=inspect_shell ubuntu /bin/bash
Unable to find image 'ubuntu' locally
Pulling repository ubuntu
86ce37374f40: Download complete
511136ea3c5a: Download complete
5bc37dc2dfba: Download complete
61cb619d86bc: Download complete
3f45ca85fedc: Download complete
78e82ee876a2: Download complete
dc07507cef42: Download complete
root@761ef6d4b28f:/#
```

首先，告诉Docker要运行docker run命令。这个命令后面是命令行标志-i和-t，前者用于打开容器的标准输入（STDIN），后者告诉Docker为容器建立一个命令行终端。这两个标志为我们和容器提供了交互shell，是创建交互型容器的基本设置。后面的--name标志为容器指定了一个名字，这是一个可选项。当没有这个选项时，Docker会为我们取一个随机的名字。接下来，我们告诉Docker使用哪个镜像去创建容器，这里使用的是ubuntu。ubuntu镜像是一个基础镜像，我们可以使用基础镜像（例如ubuntu、fedora、debian、centos等）作为创建自己镜像的基础。这里我们只是用基础镜像来启动容器，没有添加任何东西。最后，告诉Docker要在容器里面执行命令/bin/bash。

命令本身我们理解了，那么在后台会发生些什么呢？创建容器的流程如图2-1所示，当我们运行docker run命令后，Docker在本地搜索我们指定的ubuntu镜像，如果没有找到，就会到公有仓库Docker Hub中继续搜索。如果在服务器上找到了需要的镜像，Docker就会下载这个镜像，并将其保存到本地。然后，Docker使用这个镜像创建一个新的容器并将其启动；容器的文件系统是在只读的镜像文件上增加一层可读写的文件层，这样可以保证镜像不变而只记录改变的数据，这对容器的共享和传输都非常有利。接着会配置容器的网络，Docker会为容器分配一个虚拟网络接

口，并通过网桥的方式将该网络接口桥接到宿主主机上去，然后该虚拟网络接口分配一个IP地址。最后，Docker在新容器中运行指定的命令，例如我们的例子中是/bin/bash。容器创建成功后，会出现类似下面的提示符：

```
root@761ef6d4b28f:/#
```

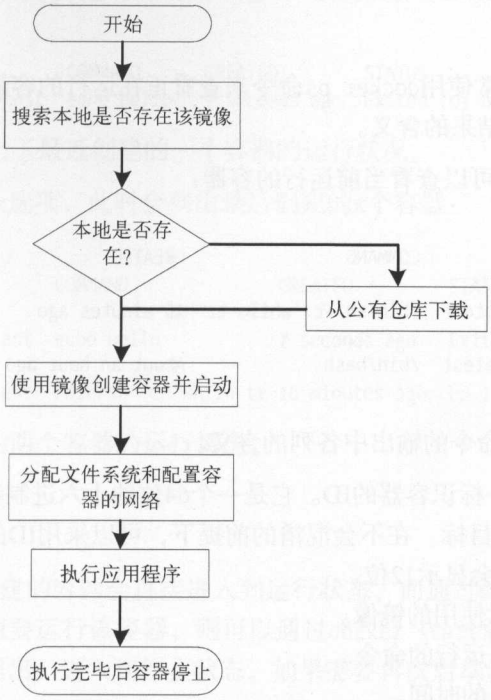


图2-1 docker run命令的内部流程

@前面的是我们在容器的登录用户root，后面的761ef6d4b28f是容器的主机名。可以使用ctrl+D或者exit命令退出该容器。容器停止并不代表容器销毁，其实容器还在，只是不再是运行态，可以通过docker ps -a命令查看到已存在的容器。

接下来，我们创建一个后台型容器。在实际的应用中，大多数容器都是后台型容器，因为服务器程序不可能因为创建容器的终端退出而退出。创建后台型容器需要使用-d参数，其创建命令如下：

```
$ sudo docker run --name daemon_while -d ubuntu /bin/sh -c    "while true; do echo hello
    world; sleep 1; done"
f40f1463221f6fdcd5ae26d223df10273b0a4831d8bf1db16d461f583ac0cfb6
```

你可能注意到了，上面的命令没有像前面的容器那样关联到一个shell，而是返回了一个容器ID后直接返回到了宿主主机的命令提示符。我们可以通过运行docker ps命令，查看新建的容器是否正在运行：

```
$ sudo docker ps
CONTAINER ID  IMAGE          COMMAND                CREATED         STATUS         PORTS  NAMES
f40f1463221f  ubuntu:latest  /bin/sh -c 'while tr  2 minutes ago   Up 2 minutes          daemon_while
761ef6d4b28f  ubuntu:latest  /bin/bash              49 minutes ago  Up 49 minutes         inspect_shell
```

可以看到，有两个容器在运行，交互型的容器inspect_shell和后台型容器daemon_while。

2.1.2　查看容器

在上一节中，我们经常使用docker ps命令来查看正在运行的容器。在这一节中，我们详细说明该命令的参数及输出结果的含义。

使用docker ps命令，可以查看当前运行的容器：

```
$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                CREATED             STATUS             PORTS
NAMES
f40f1463221f   ubuntu:latest  /bin/sh -c 'while tr  16 minutes ago      Up 16 minutes
daemon_while
761ef6d4b28f   ubuntu:latest  /bin/bash              About an hour ago   Up About an hour
inspect_shell
```

下面简要介绍一下该命令的输出中各列的含义。

- CONTAINER ID：唯一标识容器的ID。它是一个64位的十六进制数，对某个容器的操作可以通过它来标识操作目标。在不会混淆的前提下，可以采用ID的前几位来标识该容器，而在显示的时候一般会显示12位。
- IMAGE：创建容器时使用的镜像。
- COMMAND：容器最后运行的命令。
- CREATED：创建容器的时间。
- STATUS：容器的状态。如果容器是运行状态，则类似UP 49 minutes的形式，其中49分钟是容器已经运行的时间；如果容器是停止状态，则是类似Exited(0)的形式，其中数字0是容器退出的错误码，0为正常退出，其他数字则表示容器内部出现错误。
- PORTS：对外开放的端口。
- NAMES：容器名。和容器ID一样都可以唯一标识一个容器，所以同一台宿主主机上不允许有同名的容器存在，否则会提示冲突。

使用docker ps命令，只会列出当前正在运行的容器。当要查看所有的容器时，可以使用-a选项，它会告诉Docker列出所有容器，包括运行的和停止的容器。示例代码如下：

```
$ sudo docker ps -a
CONTAINER ID   IMAGE          COMMAND                CREATED             STATUS                      PORTS
NAMES
dbc5bafd5805   ubuntu:latest  echo hello             7 seconds ago       Exited (0) 6 seconds ago
hopeful_bartik
f40f1463221f   ubuntu:latest  /bin/sh -c 'while tr  16 minutes ago      Up 16 minutes
daemon_while
```

```
761ef6d4b28f   ubuntu:latest  /bin/bash          About an hour ago Up About an hour
inspect_shell
```

可以发现，使用-a选项多出了一个名称是hopeful_bartik的容器，其状态是Exited(0)，表示容器已经退出了，退出码是0。

当要查看最新创建的容器时，还可以使用-l选项，该选项告诉Docker只列出最后创建的容器：

```
$ sudo docker ps -l
CONTAINER ID   IMAGE          COMMAND      CREATED         STATUS              PORTS       NAMES
dbc5bafd5805   ubuntu:latest  echo hello   7 seconds ago   Exited (0) 6 seconds ago
```

可以看到，这里只列出了最近创建的一个容器的运行状况。

此外，还可以使用-n=x选项，此时会列出最后创建的x个容器：

```
$ sudo docker ps -n=2
CONTAINER ID  IMAGE          COMMAND              CREATED         STATUS                     PORTS
NAMES
dbc5bafd5805  ubuntu:latest  echo hello           7 seconds ago   Exited (0) 6 seconds ago
hopeful_bartik
f40f1463221f  ubuntu:latest  /bin/sh -c 'while tr 16 minutes ago  Up 16 minutes
```

这里列出了最近创建的两个容器的运行情况。

2.1.3 启动容器

通过docker run命令创建的容器会直接进入到运行状态，而通过docker create命令创建的容器则会进入到停止状态，想要运行该容器，则可以通过docker start命令来启动它。当容器运行完自己的任务后，容器会退出，进入到停止状态，如果需要再次启动该容器，可以再次用docker start命令来启动。

例如，可以通过docker start来启动之前已经停止了的inspect_shell容器：

```
$ sudo docker start inspect_shell
```

也可以使用该容器的ID来启动：

```
$ sudo docker start 761ef6d4b28f
```

容器在运行过程中，总是不可避免地会出现各种问题，严重的会导致容器因为异常而退出。而有时我们需要根据错误码来判断容器是否需要重启。默认情况下，容器是不重启的，为了让容器在退出后能够自动重启，需要用到--restart参数。--restart标志会检查容器的退出码，并据此来决定是否需要重启容器。

我们可以用下面的命令创建容器：

```
$ sudo docker run --restart=always --name docker_restart -d ubuntu /bin/sh -c "while
    true; do echo hello world; sleep 1; done"
```

在这个例子中，--restart标志被设置成always。不管容器的返回码是什么，Docker都会尝试重启容器。另外，我们也可以将其设置成on-failure，这样的话，当容器的返回值是非0时，Docker才会重启容器。on-failure标志还接受一个可选的重启次数，如下所示：

```
--restart=on-failure:5
```

表示当收到一个非0的返回码时，最多尝试重启容器5次。

2.1.4　终止容器

退出容器的方式有很多种。当容器发生严重的错误时，容器会因为异常而退出并带有错误码，这时可以通过错误码来判定容器内部发生的错误。而在正常情况下，交互型容器可以在shell中输入exit，或者是使用ctrl+d组合键来使其退出。另外，交互型容器和后台型容器都可以采用docker stop命令来停止：

```
$ sudo docker stop daemon_while
```

上述介绍的都是通过容器名来停止该容器。此外，我们还可以通过容器ID来停止容器：

```
$ sudo docker stop f40f1463221f
```

docker stop命令给容器中的进程发送SIGTERM信号，默认行为是会导致容器退出。当然，容器内程序可以捕获该信号并自行处理，例如可以选择忽略。如果要强行停止一个容器，则需要使用docker kill命令，它会给容器的进程发送SIGKILL信号，该信号将会使容器必然退出。

2.1.5　删除容器

当一个容器停止时，容器并没有消失，只是进入了停止状态，必要的话还可以重新运行。如果确定不再需要这个容器时，可以使用docker rm命令删除它：

```
$ sudo docker rm hopeful_bartik
hopeful_bartik
```

要注意一点，不可以删除一个运行中的容器，此时必须先用docker stop或docker kill命令停止它才能删除。示例如下：

```
$ docker ps
CONTAINER ID  IMAGE           COMMAND       CREATED          STATUS         PORTS  NAMES
dfc72feb607b  ubuntu:14.04    "/bin/sh"     13 seconds ago   Up 12 seconds         modest_mestorf
micall@micall-ThinkPad:~$ docker rm modest_mestorf
Error response from daemon: You cannot remove a running container. Stop the container
    before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

此时会输出错误，提示你不能够删除一个正在运行的容器，必须先停止才能够删除。当然，你也可以使用-f选项强制删除它：

```
$ docker rm -f modest_mestorf
modest_mestorf
micall@micall-ThinkPad:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

Docker并没有提供一次性删除所有容器的命令，但是可以用下面的命令来实现这个目的：

```
docker rm `docker ps -a -q`
```

这个命令会用docker ps列出当前的所有容器，-a标志列出所有容器，-q标志只列出容器的ID，不包括容器的其他信息。然后将这个列表传给docker rm命令，依次删除容器。

2.2 容器内信息获取和命令执行

在2.1节中，我们主要讲解容器的管理操作，这属于外部操作。在本节中，我们将讲解容器内部的操作，包含获取容器的内部信息以及在容器内部运行命令。

2.2.1 依附容器

依附操作attach通常用在由docker start或者docker restart启动的交互型容器中。由于docker start启动的交互型容器并没有具体终端可以依附，而容器本身是可以接收用户交互的，这时就需要通过attach命令来将终端依附到容器上。具体示例如下：

```
$ docker run -i -t ubuntu:14.04 /bin/sh
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
$ docker run -i -t --name ubuntu ubuntu:14.04 /bin/sh
# ls
bin  boot  dev etc  home  lib lib64  media  mnt  opt proc  root  run  sbin  srv  sys  tmp  usr  var
# ^C
#
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
$ docker start ubuntu
ubuntu
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
d5521fff6cdb        ubuntu:14.04        "/bin/sh"           45 seconds ago      Up 5 seconds                            ubuntu

$ docker attach ubuntu
# ls
bin  boot  dev etc  home  lib lib64  media  mnt  opt proc  root  run  sbin  srv  sys  tmp  usr  var
#
```

需要注意的是，当使用attach命令依附容器后，需要多按一次回车才会出现容器的shell交互界面。

在上面，我们特别强调的是将终端依附到交互型容器中，那么对于后台型容器，使用attach

命令会发生什么呢？我们通过实例来探索这个问题的答案。我们发现，后台型容器是无法依附终端的，因为它本身就不接受用户交互输入。

2.2.2　查看容器日志

对于交互型容器，由于本身就存在交互终端或者可以通过attach依附终端，所以查看容器的配置信息或者调试程序比较方便。但对于后台型容器，它不存在交互终端，要获取其信息，势必需要其他的方法，Docker给我们提供了logs、inspect等方法。docker logs命令用于查看容器的日志，它将输出到标准输出的数据作为日志输出到运行docker logs命令的终端上。

首先，创建一个不断输出一些内容的后台型容器:

```
$ sudo docker run -d --name deamon_logs ubuntu /bin/bash -c 'for((i=0;1;i++));do echo $i;sleep 1;done;'
956c4bb8db65aa24c2e7b865f829c1c19d2b201c9c7fc86a77d199b2589bdc54
```

deamon_logs是一个包含循环输出自然数的应用程序的容器，我们可以使用logs来查看其输出：

```
$ sudo docker logs -f  deamon_logs
0
1
2
3
...
```

可以看到，该容器的程序循环向标准输出数字0、1、2 ...。默认情况下，logs输出的是从容器启动到调用执行logs命令时的所有输出，之后的日志不再输出，并立即返回主机的控制台。如果要实时查看日志，可以使用-f标志。由于这里使用了-f标志，可以看到，日志从开始一直输出到当前时刻，并且还在不断更新，此时可以使用ctrl+C快捷键退出监视日志。

如果前面已经有很多日志，但是我们不想关心，只想要查看日志的最后部分，可以通过--tail参数。

使用--tail标志可以精确控制logs输出的日志行数。例如，查看最后5行日志：

```
$ sudo docker logs -f --tail=5 daemon_logs
723
724
725
726
727
...
```

可以看到，首先输出日志的最后5行。由于使用了-f标志，之后的日志也会不断更新出来。

为了方便调试程序，我们还可以通过-t标志查看日志产生的时刻，相关代码如下：

```
$ sudo docker logs -f --tail=5 -t daemon_logs
```

```
2014-12-27T08:16:00.873690621Z 755
2014-12-27T08:16:01.875306639Z 756
2014-12-27T08:16:02.880058459Z 757
2014-12-27T08:16:03.884377275Z 758
2014-12-27T08:16:04.885695676Z 759

...
```


2.2.3 查看容器进程

使用docker top命令，可以查看容器中正在运行的进程。

首先，创建一个后台型容器（交互型容器也行，但是要到其他控制台运行docker top）：

```
$ sudo docker run -d --name="deamon_top" ubuntu /bin/bash -c 'while true;do sleep 1; done'
d9a46de654821f15269d7427d93b3126dd3217492e7c2eb3c7ae09a18fdf8bc2
```

运行docker top命令，查看容器中的进程：

```
$ sudo docker top deamon_top
UID             PID             PPID        C       STIME
TTY             TIME            CMD
root            11699           1158        0
16:38           ?               00:00:00    /bin/bash -c while true;do sleep
1; done
root            11778           11699       0
16:39           ?               00:00:00    sleep 1
```

可以看到，这里有两个进程，一个是while循环的进程，另一个是while循环内部运行的sleep进程。

2.2.4 查看容器信息

docker inspect用于查看容器的配置信息，包含容器名、环境变量、运行命令、主机配置、网络配置和数据卷配置等：

```
$ sudo docker inspect daemon_dave
[{
    "ID": " c2c4e57c12c4c142271c031333823af95d64b20b5d607970c334784430bcbd0f ",
    "Created": "2014-05-10T11:49:01.902029966Z",
    "Path": "/bin/sh",
    "Args": [
        "-c",
        "while true; do echo hello world; sleep 1; done"
    ],
    "Config": {
        "Hostname": "c2c4e57c12c4",
    ...
```

使用-f或者--format格式化标志，可以查看指定部分的信息。

查询容器的运行状态：

```
$ sudo docker inspect --format='{{ .State.Running }}' daemon_dave
false
```

查询容器的IP地址：

```
$ sudo docker inspect --format '{{ .NetworkSettings.IPAddress }}'
daemon_dave
```

同时还可查看多个信息，例如查看容器名和运行状态：\

```
$ sudo docker inspect --format '{{.Name}} {{.State.Running}}' \
daemon_dave bob_the_container
/daemon_dave false
/bob_the_container false
```

2.2.5　容器内执行命令

在容器启动的时候，通常需要指定其需要执行的程序，然而有时候我们需要在容器运行之后中途启动另一个程序。从Docker 1.3开始，我们可以用docker exec命令在容器中运行新的任务，它可以创建两种任务：后台型任务和交互型任务。后台型任务没有用户交互终端，交互型任务具有和用户交互的输入输出终端。

让我们看一个后台型任务的例子：

```
$ sudo docker exec -d daemon_dave touch /etc/new_config_file
```

这里-d标志表示要运行一个后台型任务。接着需要指定要运行命令的容器名和要运行的命令。在这个例子里，touch命令会在daemon_dave容器中创建一个new_config_file文件。通过docker exec创建的后台任务，我们可以执行维护、监视、管理等复杂而有意义的任务。

使用docker exec命令创建交互型任务时，需要加上-t -i标志，示例如下：

```
$ sudo docker exec -t -i daemon_dave /bin/bash
```

-t和-i标志的用法与我们创建交互型容器时一样，会创建一个交互终端，并捕捉进程的标准输入和输出。通过该交互终端，我们可以在容器内运行命令和查看信息等。

2.3　容器的导入和导出

Docker的流行与它对容器的易分享和易移植密不可分。用户不仅可以把容器提交到公共服务器上，还可以将容器导出到本地文件系统中。同样，我们也可以将导出的容器重新导入到Docker运行环境中。Docker的导入和导出分别由import命令和export命令完成。

下面我们说明一下容器的导出。首先，创建一个容器：

```
$ sudo docker run -i -t --name=inspect_import ubuntu /bin/bash
root@3d2371934e2d:/#
```

然后按需要修改容器，安装需要的软件，配置系统环境。当我们完成这一切后，就可以把容器保存到本地，使用docker export命令导出容器：

```
$ sudo docker export inspect_import > my_container.tar
$ ls
my_container.tar
```

docker export 命令会把容器的文件系统以tar包的格式导出到标准输出，我们将其重定位到目标文件name.tar。将容器保存到本地文件也算是其持久化方式的一种。将容器保存到本地之后，我们就可以通过网络等方法将tar包分享给他人。

反过来，我们可以使用docker import命令导入一个本地的tar包作为镜像：

```
$ cat my_container.tar | sudo docker import - imported:container
6c16d0ec9f8ef1a1f6ef47ce92b77692dd8c5d12e669b045366bbaf71d7182b1
$ sudo docker images
REPOSITORY        TAG               IMAGE ID          CREATED             VIRTUAL SIZE
imported          container         6c16d0ec9f8e      35 seconds ago      192.5 MB
ubuntu            latest            9bd07e480c5b      3 weeks ago         192.7 MB
```

docker import会把打包的容器导入为一个镜像。

import表示从标准输入读取容器内容，我们把name.tar的内容传给了标准输入，res和tag分别代表生成的镜像和标记。

除了导入本地文件系统的tar包成为一个镜像外，我们还可以使用一个url来导入网络上的容器：

```
docker import url res:tag
```

接着就可以通过docker run命令使用导入的镜像创建我们需要的容器了。

第3章 镜 像

和容器一样，镜像是Docker的核心组件之一。镜像是容器的运行基础，容器是镜像运行后的形态，二者紧密相连又有不同。在第2章中讨论容器时，涉及镜像的地方并没有展开来讲，本章详细介绍一下镜像。在本章中，我们将详细探讨如下内容：

- 镜像的概念，其中包括镜像与容器的区别、镜像独特的组成架构等；
- 本地镜像的管理，其中将涉及获取、创建和删除等命令的使用；
- 如何通过互联网来分发我们定制的镜像。

3.1 镜像的概念

镜像是一个包含程序运行必要依赖环境和代码的只读文件，它采用分层的文件系统，将每一次改变以读写层的形式增加到原来的只读文件上。

3.1.1 镜像与容器

镜像是容器运行的基石。正如在第2章中所看到的，使用docker run命令创建一个容器并在其中运行进程时，必须指定一个镜像名称或者镜像ID。下面的命令展示了使用根镜像ubuntu创建容器并在其中运行的例子：

```
# docker run ubuntu echo "hello docker"
hello docker
```

如果将容器理解为一套程序运行的虚拟环境，那么镜像就是用来构建这个环境的模板。通过同一个镜像，我们可以构造出很多相互独立但运行环境一样的容器。

不同镜像也许有着不同的服务目标，比如ubuntu镜像用来构建一个精简的Ubuntu操作系统容器环境，wordpress镜像用来构建博客程序容器环境。此外，用户还可以在已有的任意镜像上面扩展定制出满足特殊业务需求的镜像。

3.1.2 镜像的系统结构

图3-1展示了Docker镜像的系统结构。镜像的最底层必须是一个称为启动文件系统（bootfs）的镜像，用户不会与这一层直接打交道。bootfs的上层镜像叫作根镜像（rootfs），它在通常情况下是一个操作系统，如Ubuntu、Debian 和CentOS等。用户的镜像必须构建于根镜像之上。图中所示的镜像1是通过在根镜像ubuntu上安装MySQL来创建的。在镜像1的基础上再安装一个Nginx，就又创建了镜像2。利用镜像2启动的容器里面运行的是一个已经安装好了MySQL和Nginx的Ubuntu系统。

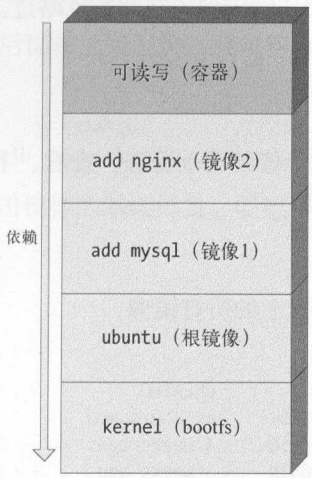


图3-1 Docker镜像的系统结构

镜像的本质是磁盘上一系列文件的集合。不难理解，创建新的镜像其实也就是对已有镜像文件进行增、删、改操作。镜像之间并不是孤立的，而是存在单向的文件依赖关系。如图3-2所示，镜像的原理与Git相似，熟悉Git工作原理的读者对此应该不会陌生。镜像1的FileA'是镜像2的FileA的修改版本，NewFile是新创建的文件，其他文件则全部引自镜像2。正因为Docker的这种文件层叠共享机制，才造就镜像占用磁盘空间小、扩展容易、传播灵活等优点。

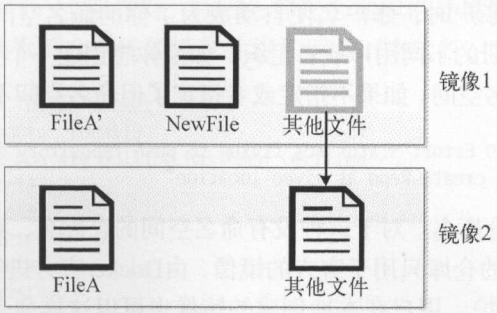


图3-2 镜像的文件依赖关系

3.1.3　镜像的写时复制机制

通过docker run命令指定镜像创建一个容器时，实际上是在该镜像之上创建一个空的可读写文件系统层级。可以将这个文件系统当成一个新的临时镜像，而命令里所指定的镜像称为父镜像。父镜像的内容都是以只读方式挂载进来的，容器会读取共享父镜像的内容。不过一旦需要修改父镜像文件，便会触发Docker从父镜像中复制这个文件到临时镜像中来，所有的修改均发生你的文件系统中，而不会对父镜像造成任何影响，这就是Docker镜像的写时复制机制。图3-2展示了这种写时复制导致的文件依赖关系。用户可以通过commit命令保存该临时镜像所做的修改，从而形成一个真正的镜像，这在3.3节中会详细讲到，在此暂且略过。

3.2　本地镜像的管理

在本节中，我们主要介绍本地镜像的基础管理：查看、下载和删除。

3.2.1　查看

通过images命令，可以列出本机上的所有镜像：

```
# docker images
REPOSITORY    TAG         IMAGE ID          CREATED             VIRTUAL SIZE
bbbb/ccccc    aaaa        1009d6e33803      45 hours ago        199.3 MB
centos        latest      ae0c2d0bdc10      4 weeks ago         224 MB
ubuntu        latest      5506de2b643b      5 weeks ago         199.3 MB
...
```

下面简要介绍一下上述代码中各个字段的含义。

- REPOSITORY：仓库名称。仓库一般用来存放同一类型的镜像，其名称由它的创建者指定（具体如何指定将在3.3节中讲到），如果创建时没有指定则为<none>。仓库的名称有下面几种形式。
  - [namespace\ubuntu]：由命名空间和实际的仓库名称组成，中间通过\隔开。当你在Docker Hub上注册一个账户时，账户名便自动成为了你的命名空间，该命名空间是用来区分Docker Hub上注册的不同用户或者组织。如果你想创建一个分发到Docker Hub上去的镜像，必须指定命名空间。如果不指定或者指定了但命名空间不符合，将会得到如下错误：

    ```
    2014/12/10 20:43:10 Error: Status 403 trying to push repository abcd/abcd: "Access Denied:
        Not allowed to create Repo at given location"
    ```

  - [ubuntu]：只有仓库名。对于这种没有命名空间的仓库名，我们可以认为它属于顶级命名空间。该空间的仓库只用于官方的镜像，由Docker官方进行管理，但一般会授权给第三方进行开发维护。用户在本地创建的镜像也可以这样命名，但是无法分发到Docker Hub上进行共享。

- [dl.dockerpool.com:5000\ubuntu:12.04]：指定URL路径的方式。如果该镜像不是放置在Docker Hub上，而是放置在你自己搭建的Hub或者第三方Hub上，则使用这种方式命名。dl.dockerpool.com:8080是第三方Hub的主机名及端口，ubuntu是镜像名称。

- TAG：用于区分同一仓库中的不同镜像。如果未指定，默认为latest。
- IMAGE ID：每个镜像都有一个字符串类型、长为64位的HashID，用来全网标识一个镜像。该字段只展示前面一部分，因为这一部分已经足以在本机唯一标识一个镜像了。
- CREATED：镜像的创建时间。
- VIRTUAL SIZE：镜像所占用的虚拟大小，该大小包含了所有共享文件的大小。

我们还可以通过在images命令后面添加通配符，找出符合条件的一系列镜像：

```
# docker images ub*
REPOSITORY  TAG       IMAGE ID       CREATED         VIRTUAL SIZE
ubuntu      latest    5506de2b643b   5 weeks ago     199.3 MB
```

使用images命令，一般只会列出镜像的基础信息。要想得到一个镜像更详细的信息，可以通过inspect命令：

```
# docker inspect ubuntu
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/bash"
        ],
...
```

3.2.2 下载

使用docker run命令运行一个镜像时，Docker首先会在本机寻找该镜像，如果本机不存在，会继续去Docker Hub上面搜索符合条件的镜像并将其下载下来运行，相关代码如下：

```
# docker run ubuntu echo "hello docker"
Unable to find image 'ubuntu' locally
ubuntu:latest: The image you are pulling has been verified
01bf15a18638: Pull complete
30541f8f3062: Pull complete
e1cdf371fbde: Pull complete
9bd07e480c5b: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for ubuntu:latest
hello docker
```

这里需要说明的是，Docker Hub上存在许多由各用户和组织创建的镜像，对于有些已经存在的镜像，可以直接下载下来使用，没有必要再去重复发明轮子。

通过search命令，可以在Docker Hub上搜索符合要求的镜像：

```
# docker search wordpress
NAME              DESCRIPTION                     STARS     OFFICIAL   AUTOMATED
wordpress         The WordPress...                138       [OK]
tutum/wordpress Wordpress Docker image -  29        [OK]
...
```

下面简要介绍一下上述代码中各个字段的含义。

- NAME：镜像的名称。由命名空间和仓库名构成，如果没有命名空间，说明该镜像属于Docker Hub的官方镜像。
- DESCRIPTION：镜像的简要描述，创建者可以登录Docker Hub修改该项。
- STARS：用户对镜像的评分。评分越高，说明质量越高。
- OFFICIAL：是否为官方镜像。一般情况下，官方镜像更可靠、更稳定。
- AUTOMATED：是否使用了自动构建，这将在3.4.3节中介绍。

为了在运行镜像时不用再费时等待下载镜像，可以通过pull命令预先将镜像拉回到本地。镜像名必须完整地包含命名空间和仓库名。如果一个仓库中存在多个镜像，还必须指定TAG，否则使用默认的TAG——latest。

例如，下面直接运行容器ubuntu，我们发现本地没有，于是Docker向Docker Hub中拉取：

```
# docker run ubuntu echo "hello docker"
Unable to find image 'ubuntu' locally
ubuntu:latest: The image you are pulling has been verified
01bf15a18638: Pull complete
30541f8f3062: Pull complete
e1cdf371fbde: Pull complete
9bd07e480c5b: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for ubuntu:latest
hello docker
```

这时我们就可以预先通过pull命令来完成拉去的工作，而不用等到要运行的时候，这样可以节省运行时的等待时间：

```
# docker pull ubuntu
Unable to find image 'ubuntu' locally
ubuntu:latest: The image you are pulling has been verified
01bf15a18638: Pull complete
30541f8f3062: Pull complete
e1cdf371fbde: Pull complete
9bd07e480c5b: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for ubuntu:latest
```

3.2.3 删除

对于那些不再需要的镜像，可以使用rmi命令删除。与移除容器的命令rm相比，删除镜像的命令多了一个i，i即image的意思，这非常容易记住。示例代码如下：

```
# docker rmi c20fd090cbb6
Deleted: c20fd090cbb692b556e5910e7c1092ce292fc5efc7493deaecfec5d746c3cf42
Deleted: 6d3f47df05998b84fbd5ec433785851f64d2d0fc4c28cd6b0f4a050c7e360f8a
# docker rmi jamtur01/static_web
Deleted: c20fd090cbb692b556e5910e7c1092ce292fc5efc7493deaecfec5d746c3cf42
Deleted: 6d3f47df05998b84fbd5ec433785851f64d2d0fc4c28cd6b0f4a050c7e360f8a
```

在rmi后面，可以指定一个或多个镜像名称或者镜像ID，其中多个镜像之间使用空格隔开。如果不指定，则会删除TAG为latest的镜像。

有时候，会遇到镜像删不掉的情况，一般出现这个问题的原因是这个镜像被容器所依赖。即便容器已经停止运行了，也仍然需要依赖镜像。用户可以使用-f参数进行强制删除，或者选择另一种相对温柔的办法，那就是先将依赖它的镜像和容器移除。

下面的例子演示了存在容器而导致镜像删除失败的情况：

```
# docker rmi ubuntu
Error response from daemon: Conflict, cannot delete 9bd07e480c5b because the container 1ea331e3ce1e
    is using it, use -f to force
2014/12/10 20:11:09 Error: failed to remove one or more images
```

如果本地有很多已经停止运行的容器，一个个删除很麻烦，此时可以通过下面的命令将这些容器一次性删除掉，这样就能减少无用容器对镜像的依赖：

```
# docker rm $(docker ps -a -q)
e13788bcce4c
08530fb59349
9a0ab77d4449
```

其中docker ps -a -q命令用来列出所有容器的ID。

3.3 创建本地镜像

在2.3节中，我们知道可以将一个本地的tar包导入为镜像，其前提是该tar包是由镜像导出的。不管怎样，将tar包导入也算是创建本地镜像的一种方法。接下来，我们介绍另外两种创建本地镜像的方法。

3.3.1 使用 commit 命令创建本地镜像

正如3.1节所讲，使用镜像创建并运行一个容器，实际上是在父镜像的基础上创建一个可读写的文件层级。我们在容器里所做的修改（包括安装新的应用程序、更改系统配置），都发生在

这个层级上面。下面的一系列命令展示了在ubuntu镜像上创建和运行一个容器，并在该容器上安装SQLite3以及在根目录下创建一个名为hellodocker的文件，并且在这个文件中写入test docker commit：

```
# docker run -t -i ubuntu
root@0ddf83b837fe:/# apt-get update
.....(apt-get update命令的输出，为节省篇幅，在此省略)
root@0ddf83b837fe:/# apt-get install sqlite3
.....(apt-get install sqlite3命令的输出)
root@0ddf83b837fe:/# echo "test docker commit" >> hellodocker
root@0ddf83b837fe:/# exit
```

这里创建的容器的ID是0ddf83b837fe，这个ID在使用commit命令时会用到。在容器中完成修改之后，使用exit命令安全退出容器。

接下来，我们使用commit命令将容器里的所有修改提交到本地库中，形成一个全新的镜像：

```
# docker commit -m="Message" --author="XIXIHE" 0ddf83b837fe xixihe/sqlite3:v1
a0345b9244e19f03c3456b29e488b653a9cd7b9b1686e4d60a04ecb802247c95
# docker images
REPOSITORY        TAG     IMAGE ID      CREATED              VIRTUAL SIZE
xixihe/sqlite3    V1      a0345b9244e1  About a minute ago   220.9 MB
...(略)
```

成功执行commit命令后，会返回一个长字符串，这个字符串就是刚创建的镜像的完整ID。commit命令后跟随的0ddf83b837fe参数是我们刚才作出修改的容器ID，这个ID也可以通过docker ps -l -q（用于获取最近创建的容器ID）命令得到。-m参数是描述我们此次创建image的信息，--author参数用来指定作者信息，xixihe和sqlite3分别是仓库名和镜像名，v1是TAG名。

接下来，我们使用刚才创建的镜像来构建一个容器并运行，以检视所做的修改：

```
# docker run -t -i xixihe/sqlite3:v1
root@ce066126f750:/# sqlite3 -version
3.8.2 2013-12-06 14:53:30 27392118af4c38c5203a04b8013e1afdb1cebd0d
root@ce066126f750:/# cat hellodocker
test docker commit
```

从以上命令的反馈来看，SQLite3已经成功安装了，同时根目录下的hellodocker文件也存在。

3.3.2　使用 Dockerfile 创建镜像

与第一种方法相比，我们更推荐使用Dockerfile来构建镜像。将需要对镜像进行的操作全部写到一个文件中，然后使用docker build 命令从这个文件中创建镜像。这种方法可以使镜像的创建变得透明和独立化，并且创建过程可以被重复执行。Dockerfile文件以行为单位，行首为Dockerfile命令，命令都是大写形式，其后紧跟着的是命令的参数。

下面是一个Dockerfile文件实例，本例子并没有实际意义，只是为了将知识点都覆盖到：

```
# Version: 1.0.1
```

```
FROM ubuntu:latest

MAINTAINER xxh "xxh@qq.com"

#设置root用户为后续命令的执行者
USER root

#执行操作
RUN apt-get update
RUN apt-get install -y nginx

#使用&&拼接命令
RUN touch test.txt && echo "abc" >> abc.txt

#对外暴露端口
EXPOSE 80 8080 1038

#添加文件
ADD abc.txt /opt/

#添加文件夹
ADD /webapp /opt/webapp

#添加网络文件
ADD https://www.baidu.com/img/bd_logo1.png /opt/

#设置环境变量
ENV WEBAPP_PORT=9090

#设置工作目录
WORKDIR /opt/

#设置启动命令
ENTRYPOINT ["ls"]

#设置启动参数
CMD ["-a", "-l" ]

#设置卷
VOLUME ["/data", "/var/www"]

#设置子镜像的触发操作
ONBUILD ADD . /app/src
ONBUILD RUN echo "on build excuted" >> onbuild.txt
```

下面介绍一下上述代码中各个命令的含义。

- FROM：指定待扩展的父级镜像。除了注释外，在文件开头必须是一个FROM指令，接下来的指令便在这个父级镜像的环境中运行，直到遇到下一个FROM指令。通过添加多个FROM命令，可以在同一个Dockerfile文件中创建多个镜像。
- MAINTAINER：用来声明创建的镜像的作者信息。在上述代码中，xxh是用户名，xxh@qq.com是邮箱。这个命令并不是必需的。

- RUN：用来修改镜像的命令，常用来安装库、程序以及配置程序。一条RUN指令执行完毕后，会在当前镜像上创建一个新的镜像层，接下来的指令会在新的镜像上继续执行。RUN语句又有两种形式：

```
RUN apt-get update
RUN [ "apt-get", "update" ]
```

  第一种形式是在/bin/sh环境中执行指定的命令，第二种形式是直接使用系统调用exec来执行。我们还可以使用&&符号将多条命令连接在同一条RUN语句中执行：

```
RUN apt-get update && apt-get install nginx。
```

- EXPOSE：用来指明容器内进程对外开放的端口，多个端口之间使用空格隔开。运行容器时，通过参数-P（大写）即可将EXPOSE里所指定的端口映射到主机上另外的随机端口，其他容器或主机就可以通过映射后的端口与此容器通信。同时，我们也可以通过-p（小写）参数将Dockerfile中EXPOSE中没有列出的端口设置成公开的。
- ADD：向新镜像中添加文件，这个文件可以是一个主机文件，也可以是一个网络文件，也可以是一个文件夹。

  ADD命令的第一个参数用来指定源文件（夹），它可以是文件路径、文件夹的路径或网络文件的URL地址。需要特别注意的是，如果是文件路径或文件夹路径，它必须是相对Dockerfile所在目录的相对路径。如果是一个文件URL，在创建镜像时，会先下载下来，然后再添加到镜像里去。第二个参数是文件需要放置在目标镜像的位置。如果源文件是gzip、bzip2或者xz形式的压缩文件，Docker会先解压缩，然后将文件添加到镜像的指定位置。如果源文件是一个通过URL指定的网络压缩文件，则不会解压。

- VOLUME：该命令会在镜像里创建一个指定路径（文件或文件夹）的挂载点，这个路径可以来自主机或者其他容器。多个容器可以通过同一个挂载点共享数据，即便其中一个容器已经停止，挂载点也仍然可以访问，只有当挂载点的容器引用全部消失时，挂载点才会自动删除。关于卷的概念，我们会在第4章中详细介绍。
- WORKDIR：为接下来执行的指令指定一个新的工作目录，这个目录可以是绝对目录，也可是相对目录。根据需要，WORKDIR可以被多次指定。当启动一个容器时，最后一条WORKDIR指令所指的目录将作为容器运行的当前工作目录。
- ENV：设置容器运行的环境变量。在运行容器的时候，通过-e参数可以修改这个环境变量值，也可以添加新的环境变量：

```
# docker run -e WEBAPP_PORT=8000 -e WEBAPP_HOST=www.example.com ...
```

- CMD：用来设置启动容器时默认运行的命令。假如Dockerfile中CMD命令是这样的：

```
CMD [ "ls", "-a", "-l" ]
```

  那么运行容器的效果如下：

```
# docker run xixihe/test
total 72
drwxr-xr-x  44 root root 4096 Dec 11 06:07 .
drwxr-xr-x  44 root root 4096 Dec 11 06:07 ..
-rwxr-xr-x   1 root root    0 Dec 11 06:07 .dockerenv
-rwxr-xr-x   1 root root    0 Dec 11 06:07 .dockerinit
drwxr-xr-x   2 root root 4096 Dec  4 00:16 bin
drwxr-xr-x   2 root root 4096 Apr 10  2014 boot
...
```

这样启动一个容器时，就不再需要指定运行的程序或命令了。当然，我们仍然可以重新指定启动命令以覆盖在Dockerfile文件中指定的命令：

```
# docker run xixihe/test echo "hello docker"
hello docker
```

CMD参数的格式和RUN类似，也有两种形式，并且两者达到的结果是一样的：

```
CMD ls -l -a
CMD [ "ls", "-l", "-a" ]
```

- ENTRYPOINT：与CMD类似，它也是用来指定容器启动时默认运行的命令。

假如，我们指定的ENTRYPOINT是这样的：

```
ENTRYPOINT [ "ls", "-l"]
```

并构建出名为xixihe/newImage的镜像，那么# docker run xixihe/newImage的运行结果与# docker run ubuntu ls -l一样，而# docker run xixihe/newImage -a的运行结果与# docker run ubuntu ls -l -a一样。

不难发现，ENTRYPINT和CMD的区别在于运行容器时添加在镜像名之后的参数，对ENTRYPOINT是拼接，而对于CMD命令则是覆盖。幸运的是，我们在运行容器的时候可以通过--entrypoint来覆盖Dockerfile中的指定：

```
# docker run --entrypoint echo xixihe/newImage "hello docker"
hello docker
```

通常情况下，我们会将CMD和ENTRYPOINT搭配起来使用。ENTRYPOINT用于指定需要运行的命令，CMD用于指定运行命令所需要的参数，示例如下：

```
ENTRYPOINT [ "ls" ]
CMD [ "-a", "-l" ]
```

- USER：为容器的运行及接下来RUN、CMD、ENTRYPOINT等指令的运行指定用户或UID。
- ONBUILD：触发器指令。构建镜像的时候，Docker的镜像构建器会将所有的ONBUILD指令指定的命令保存到镜像的元数据中，这些命令在当前镜像的构建过程中并不会执行。只有新的镜像使用FROM指令指定父镜像为这个镜像时，便会触发执行。

下面的两条ONBUILD命令表明，使用FROM以这个Dockerfile构建出的镜像为父镜像，构建子镜像时将自动执行ADD . /app/src和RUN echo "on build excuted" >> onbuild.txt这两个操作：

```
ONBUILD ADD . /app/src
ONBUILD RUN echo "on build excuted" >> onbuild.txt
```

接下来，我们使用build命令来构建镜像：

```
# docker build -t xixihe/test:v1 .
Sending build context to Docker daemon 6.656 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
 ---> 8eaa4ff06b53
Step 1 : MAINTAINER xxh "xxh@qq.com"
 ---> Using cache
 ---> f9caa95a4e54
Step 2 : USER root
 ---> Using cache
 ---> 05ee079f4925
Step 3 : RUN apt-get update
 ---> Using cache
 ---> 73ede9cd2370
Step 4 : RUN apt-get install -y nginx
 ---> Running in 08e9129ef019
Reading package lists...
Building dependency tree...
Reading state information...
...(略)
 ---> eed9121c42e0
Removing intermediate container 08e9129ef019
Step 5 : RUN touch test.txt && echo "abc" >> abc.txt
 ---> Running in 141802f35d94
 ---> d3625958dc52
Removing intermediate container 141802f35d94
Step 6 : EXPOSE 80 8080 1038
 ---> Running in f6416dc4a06a
 ---> 0b49809bb5bd
Removing intermediate container f6416dc4a06a
Step 7 : ADD abc.txt /opt/
 ---> d59504f2e153
Removing intermediate container f894b8dbe851
Step 8 : ADD /webapp /opt/webapp
 ---> 29c74c759648
Removing intermediate container f1743fe68cd2
Step 9 : ADD https://www.baidu.com/img/bd_logo1.png /opt/
Downloading [==================================================>] 7.877 kB/7.877 kB
 ---> fadd6b26b530
Removing intermediate container dd7d02402e56
Step 10 : ENV WEBAPP_PORT 9090
 ---> Running in 790a3a67287f
 ---> cec6fc5d3142
Removing intermediate container 790a3a67287f
Step 11 : WORKDIR /opt/
```

```
 ---> Running in d212a6e972d9
 ---> 124b42584b26
Removing intermediate container d212a6e972d9
Step 12 : ENTRYPOINT ls
 ---> Running in 90f8af6462f8
 ---> cc2d6e77e0a5
Removing intermediate container 90f8af6462f8
Step 13 : CMD -a -l
 ---> Running in f7599cc842c6
 ---> 8daa393de892
Removing intermediate container f7599cc842c6
Step 14 : VOLUME /data /var/www
 ---> Running in 94c93a0b4dd4
 ---> d8f925114884
Removing intermediate container 94c93a0b4dd4
Step 15 : ONBUILD add . /app/src
 ---> Running in 3a3275168c41
 ---> 642e0f8ad100
Removing intermediate container 3a3275168c41
Step 16 : ONBUILD run echo "on build excuted" >> onbuild.txt
 ---> Running in 4c54d1262b21
 ---> dd2d6ebe0215
Removing intermediate container 4c54d1262b21
Successfully built dd2d6ebe0215
```

其中-t参数用来指定镜像的命名空间、仓库名及TAG。这个值可以在镜像创建成功之后通过tag命令修改，事实上是创建一个镜像的两个名称引用，如下所示的xixihe/test:v1和xixihe/test:v2指向的是同一个镜像实体8758374dc545：

```
# docker tag xixihe/test:v1 xixihe/test:v2
# docker images
REPOSITORY      TAG     IMAGE ID        CREATED          VIRTUAL SIZE
xixihe/test     v2      8758374dc545    5 minutes ago    192.7 MB
xixihe/test     v1      8758374dc545    5 minutes ago    192.7 MB
ubuntu          latest  9bd07e480c5b    6 days ago       192.7 MB
```

紧跟-t参数的是Dockerfile文件所在的相对目录，本例使用的是当前目录，即“.”。从以上的构建输出中可以发现，构建的过程是分步骤的，每一条指令相当于构建一个临时镜像，直到最后一步生成我们的目标镜像。下面以Step 5为例进行介绍：

```
Step 5 : RUN touch test.txt && echo "abc" >> abc.txt
 ---> Running in 141802f35d94
 ---> d3625958dc52
 Removing intermediate container 141802f35d94
```

它从前一个临时镜像创建出容器141802f35d94，然后在这个容器中执行RUN touch test.txt && echo "abc" >> abc.txt命令，接着提交这个容器为一个临时镜像，供下一条Dockerfile指令使用，最后将创建的临时容器141802f35d94删除。通过设置docker build命令参数-rm=false，可以避免临时缓存被删除。

另外，Step 0、Step 1、Step 2和Step 3与其他步骤却有所不同：

```
Step 0 : FROM ubuntu:latest
 ---> 8eaa4ff06b53
Step 1 : MAINTAINER xxh "xxh@qq.com"
 ---> Using cache
 ---> f9caa95a4e54
Step 2 : USER root
 ---> Using cache
 ---> 05ee079f4925
Step 3 : RUN apt-get update
 ---> Using cache
 ---> 73ede9cd2370
```

这是由Docker构建器的缓存机制所致，每条指令执行都会产生一个缓存镜像，如果我们的指令链执行过了且产生了缓存镜像，那么下一次再执行这条指令链时，就可以直接使用缓存镜像，而无须重新执行指令链一遍。例子中的8eaa4ff06b53、f9caa95a4e54、05ee079f4925和73ede9cd2370即已经存在的缓存镜像的ID。通过设置docker build命令参数--no-cache=true，可以禁用缓存机制。

接下来，我们就可以使用刚才构建的镜像来创建一个容器了。在容器里，读者可自行检视Dockerfile中的操作是否已经生效，在此不再展开。

除了使用本地Dockerfile文件外，我们还可以通过指定一个Git仓库来构建。待构建的Dockerfile文件需要放置在仓库的根目录下，对应的Dockerfile文件里的ADD命令所依赖的文件也必须放置在Git仓库目录中。在使用build命令时，Docker会自动将文件下载到本地镜像中来。需要注意的是，build命令所需要的Git地址形式与GitHub上复制到的地址有所不同，如果直接使用来自GitHub的地址，将会报错。

GitHub上复制的地址形式：git@github.com:xixihe/gitDockerFile.git。

build命令所需要的地址形式：git://github.com/xixihe/gitDockerFile.git。示例代码如下：

```
# docker build -t xixihe/test:v1 git://github.com/xixihe/gitDockerFile.git
Sending build context to Docker daemon 49.15 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> 9bd07e480c5b
Step 1 : ADD README.md /opt/
 ---> 66ef4ce82a4b
Removing intermediate container 95e82db31db1
Successfully built 66ef4ce82a4b
```

3.4　Docker Hub

在本节中，我们将讲解Docker Hub相关的内容，包括基本介绍、分发本地镜像到Docker Hub、自动化构建以及创建私有库。

3.4.1 Docker Hub 简介

Docker Hub的网址是https://hub.docker.com，它与提供源代码托管服务的GitHub类似，不同的是Docker Hub提供的是镜像托管服务。利用Docker Hub，我们可以搜索、创建、分享和管理镜像，还可以利用其提供的自动化构建技术直接在集群云服务器上构建镜像。

Docker Hub为用户提供不限数目的公开镜像托管服务，但仅提供一个私有镜像托管服务。如果需要更多的私有镜像托管，需要额外付费。

Docker Hub上的镜像分为两类。一类是官方镜像，比如ubuntu、nginx、redis、mysql、wordpress等，此类镜像一般由权威的第三方（比如Canonical、Oracle、Red Hat等极具背景的大公司）进行开发维护，最后还需要Docker官方认证通过。另一类为普通用户镜像。

3.4.2 镜像的分发

想要将本机上创建的镜像分发到互联网供其他用户使用，最便捷的方式就是使用Docker Hub。首先登录Docker Hub官网注册，见图3-3。

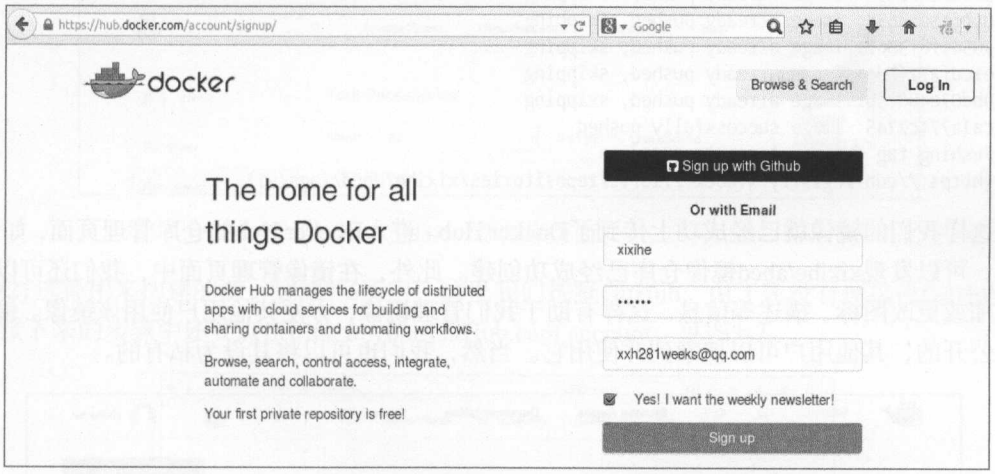


图3-3 Docker注册页面

如果你已经拥有GitHub账号，可以点击用户名框上的Sign up with Github，使用GitHub账号直接登录。注册时我们所填的用户名就是我们在Docker Hub上的命名空间，在此我们使用的是xixihe，即笔者的用户名。最后，需要填入一个有效的邮箱。注册完后，需要使用此邮箱收取激活你的Docker Hub账号的邮件。

注册成功之后，在命令行客户端登录我们刚才注册的账号：

```
# docker login
Username: xixihe
```

```
Password:
Email: xxh281weeks@qq.com
Login Succeeded
```

登录信息都保存在~/.dockercfg文件中：

```
# cat ~/.dockercfg
{
    "https://index.docker.io/v1/":{"auth":"eGl4aWhlOndob2FtaQ==","email":"xxh281weeks@qq.com"}
}
```

用户名和密码通过哈希运算之后保存在auth字段，这样可以保证密码的安全性。当然，我们也可以在首次上传镜像时由Docker主动提示我们输入密码。

登录成功之后，使用push命令上传镜像。如果不指定镜像TAG，指定的仓库在本地的所有镜像都会上传到Docker Hub。下面的push命令将3.2节创建的镜像上传到Docker Hub：

```
# docker push xixihe/abcd:v1
The push refers to a repository [xixihe/abcd] (len: 1)
Sending image list
Pushing repository xixihe/abcd (1 tags)
511136ea3c5a: Image already pushed, skipping
01bf15a18638: Image already pushed, skipping
30541f8f3062: Image already pushed, skipping
e1cdf371fbde: Image already pushed, skipping
9bd07e480c5b: Image already pushed, skipping
ca1a774c2745: Image successfully pushed
Pushing tag for rev [ca1a774c2745] on
{https://cdn-registry-1.docker.io/v1/repositories/xixihe/abcd/tags/v1}
```

这样我们的镜像就已经成功上传到了Docker Hub。进入Docker Hub的仓库管理页面，如图3-4所示，可以发现xixihe/abcd镜像仓库已经成功创建。此外，在镜像管理页面中，我们还可以为镜像添加或更改图标、描述等信息，这样有助于我们管理镜像，方便其他用户使用该镜像。镜像默认是公开的，其他用户可以搜索到并使用它。当然，我们也可以将其设为私有的。

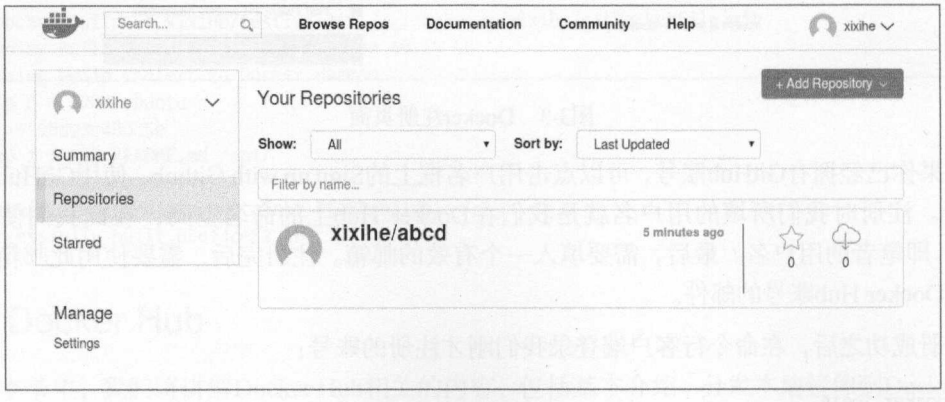


图3-4　仓库管理页面

3.4.3 自动化构建

除了在本地创建镜像然后使用push命令将其推送到Docker Hub之外，我们还可以使用Docker Hub提供的自动化构建技术在服务端直接构建镜像。通过在Docker Hub连接一个包含Dockerfile文件的Git Hub或Bit Bucket的仓库，Docker Hub的构建集群服务器就会自动构建镜像。通过这种方式构建出来的镜像会被标记为Automated Build，也可以称为受信构建（Trusted Build）。

使用自动化构建有以下几个优点。

- 用户可以确保他拉取的镜像是使用特定方式构建出来的。
- 访问你的Docker Hub的用户能够自由查阅Dockerfile文件。
- 因为构建的过程是自动的，所以能确保仓库里的镜像都是最新的。

下面我们将一步一步地演示如何使用GitHub来自动构建镜像。

(1) 登录到Docker Hub，在我的镜像页面点击右上角的Add Repository，见图3-5，然后选择下面的Automated Build。

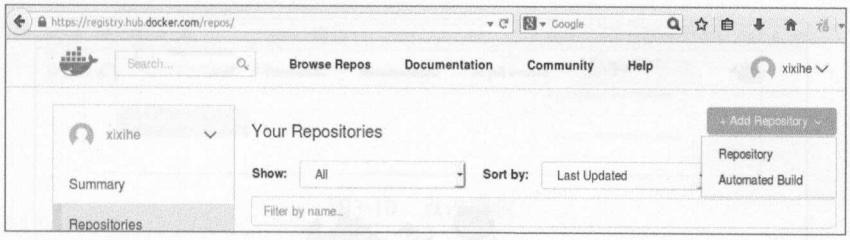


图3-5 添加库页面

(2) 选择用来构建的源，如图3-6所示，这里我们选择GitHub。由于还没有连接过GitHub，所以在接下来的步骤中还要点击Link to your GitHub.com account，见图3-7。

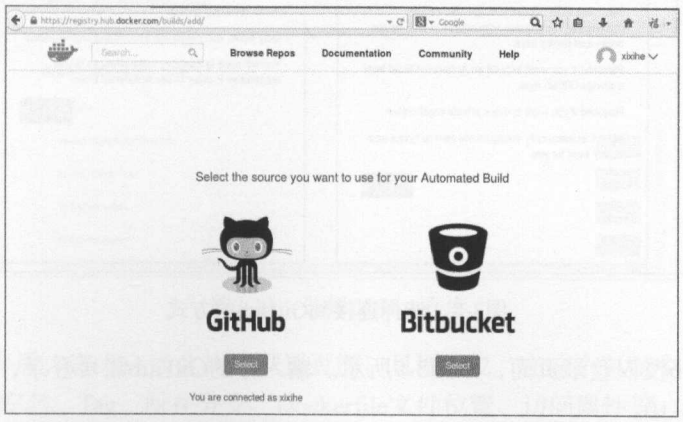


图3-6 选择构建源页面

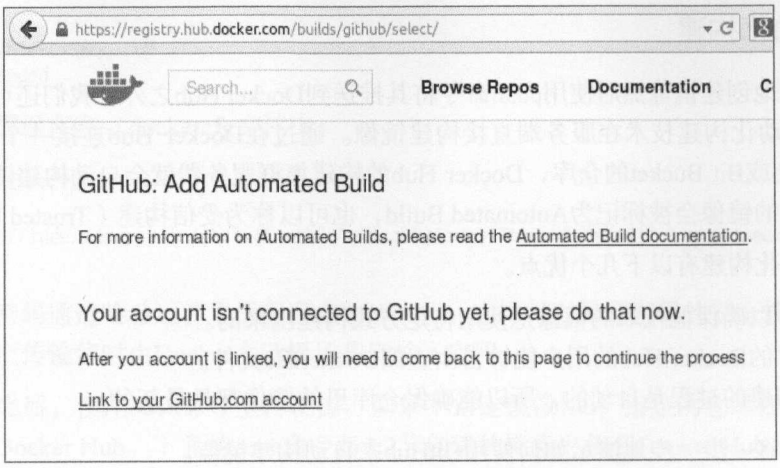


图3-7　第一次连接GitHub

(3) 选择连接到GitHub的方式，见图3-8，这里我们选择第一种：Public and Private(recommended)。

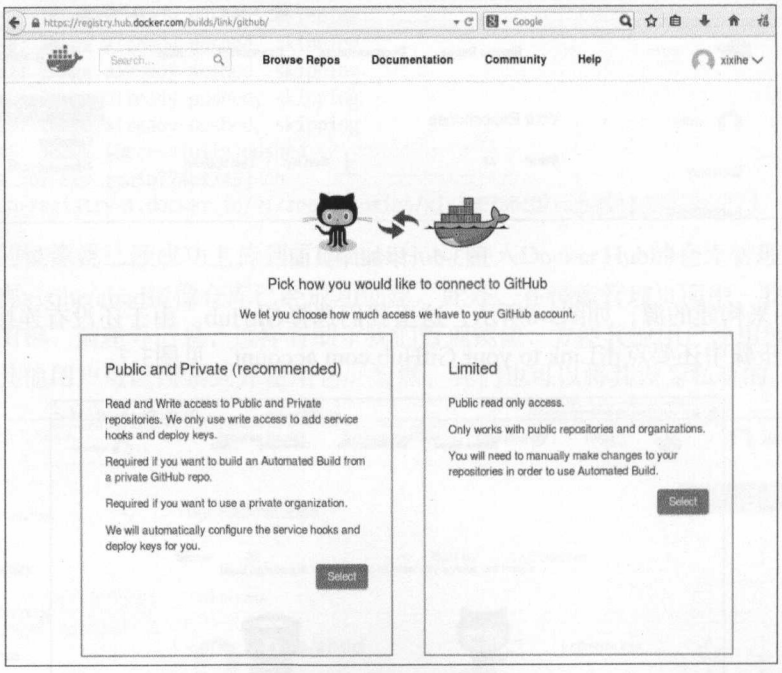


图3-8　选择连接到GitHub的方式

(4) 进入GitHub授权登录页面，如图3-9所示。输入你的GitHub账号登录，然后选择Authorize application，见图3-10。

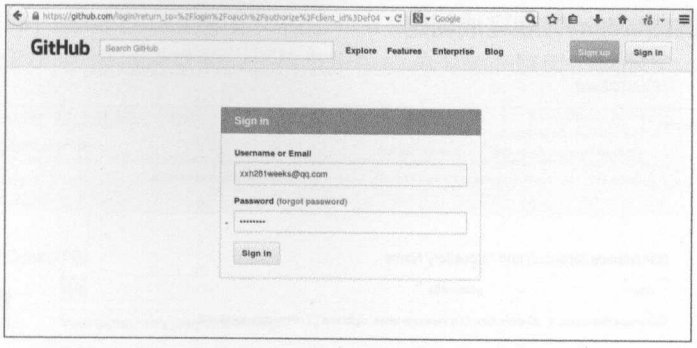


图3-9 GitHub授权登录页面

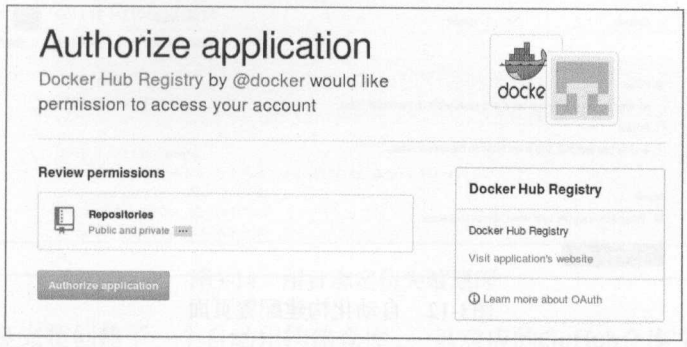


图3-10 点击授权

(5) 成功授权后，会进入GitHub仓库选择页面，见图3-11，从中选择你想要构建的仓库。

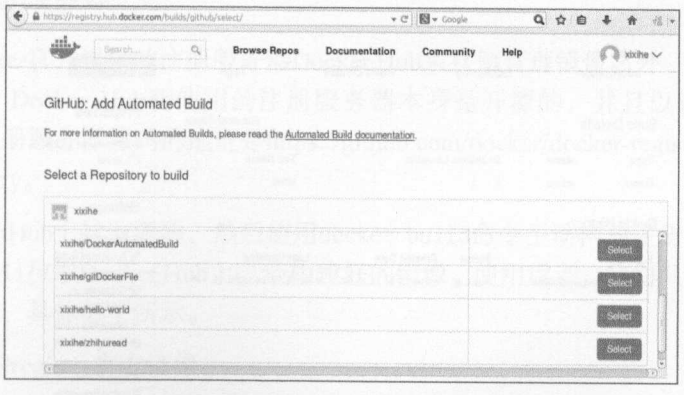


图3-11 选择GitHub仓库

(6) 选择完仓库后，进入自动化构建配置页面，见图3-12。在这个页面中，我们可以配置镜像的命名空间、仓库名、Tag、所在分支、Dockerfile文件位置、访问属性等。当然，我们也可以全部使用默认值。

(7) 点击Create Repository按钮完成创建。

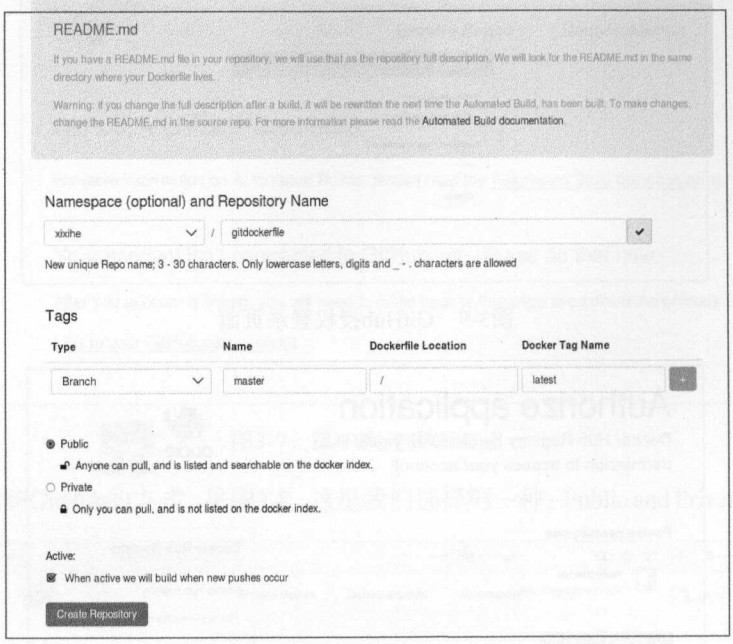


图3-12　自动化构建配置页面

(8) 进入自动化构建详情页面，见图3-13。

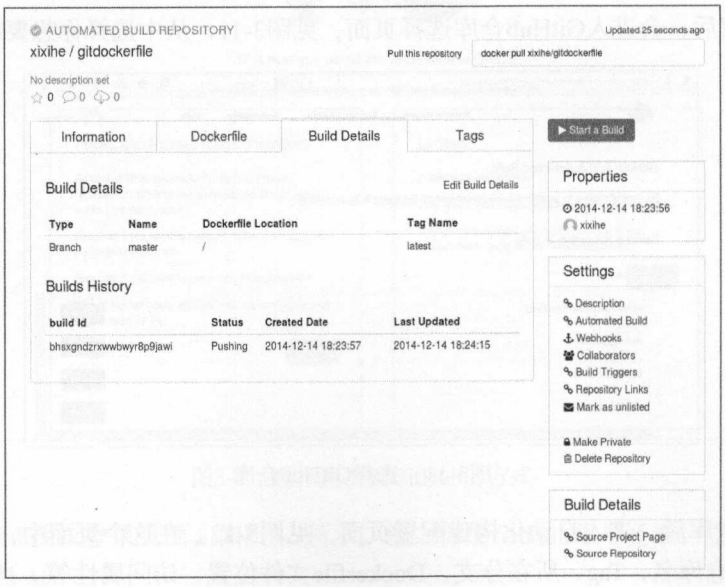


图3-13　自动化构建详情页面

(9) 通过步骤(8)中Build Details选项卡中的build Id链接可以跳转到镜像构建的过程信息页面。如果构建失败，我们可以通过这里的Logs信息定位构建失败的原因，见图3-14。

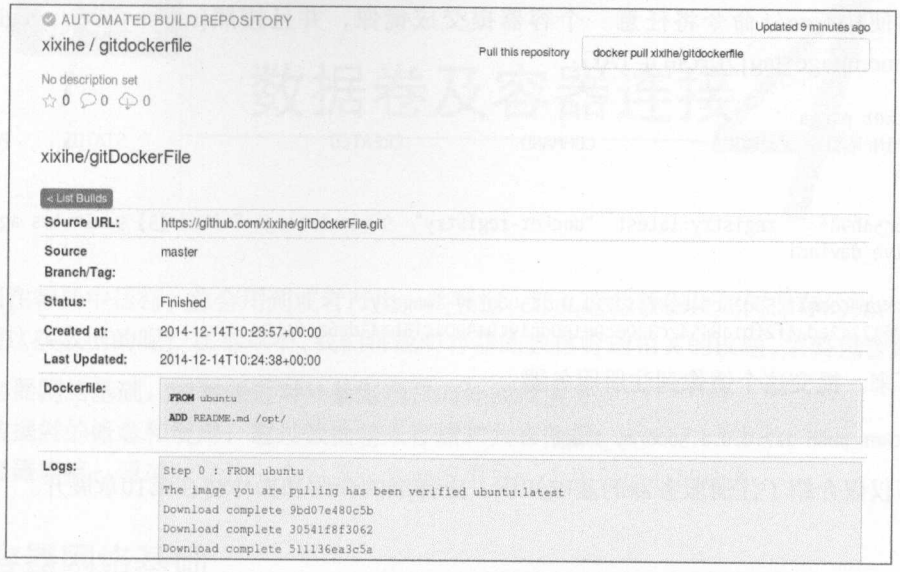


图3-14 用日志定位失败原因

至此，我们已经完整创建了一个自动化构建仓库。一旦对应的GitHub仓库文件有更新，Docker Hub上的镜像构建就会被自动触发，从而保证镜像都是最新的。

3.4.4 创建注册服务器

除了使用Docker官方提供的注册服务器Docker Hub来存储管理镜像之外，我们还可以搭建自己的注册服务器。Docker Hub所使用的注册服务器本身是开源的，并且以镜像的形式分发在Docker Hub上。注册服务器源码的地址是https://github.com/docker/docker-registry，Docker Hub上的镜像名是registry。

我们可以从GitHub上拉取源码，然后使用docker build命令手动构建注册服务器的镜像，也可以使用docker pull拉取Docker Hub上已经构建好的镜像。使用后者，我们只需要两步就能完成注册服务器的创建，具体如下所示。

(1) 拉取最新的registry官方镜像：

```
# docker pull registry
```

(2) 运行registry：

```
# docker run -p 5000:5000 -d -i -t registry
```

这样我们的注册服务器就已经成功地在5000端口运行了。接下来，可以将我们的镜像提交到这个注册服务器上。

我们使用commit命令将任意一个容器提交成镜像，并且按照[registry_host: registry_port\image_name:image_tag]方式指定TAG：

```
# docker ps -a
CONTAINER ID    IMAGE            COMMAND          CREATED                    STATUS     PORTS
NAMES
...
56d26c54b98f    registry:latest  "docker-registry"  9 minutes ago Exited (3) 9 minutes ago
pensive_davinci
...
# docker commit 56d26c54b98f 127.0.0.1:5000/my_image:v1
e35c26122c2ada4f2fb1a84542c3a0ec9e1e0dc191949e01cf5ba43da6aef410
```

接下来，提交这个镜像到注册服务器：

```
# docker push 127.0.0.1:5000/my_image:v1
```

本节仅仅介绍了注册服务器的基础知识，更详细的介绍和操作将在第10章展开。

第4章

数据卷及容器连接

应用在容器中运行，总会用到或者产生一些数据，那么这些数据是如何保存的呢？外部又是如何使用这些数据的呢？在本章中，我们将说明容器的数据管理相关的议题，主要包含如下内容。

- **容器网络基础**：容器通过对外暴露端口向外提供服务。
- **数据卷的概念和使用**：通过数据卷来存储和共享数据。
- **容器连接**：通过互联让一个容器安全地使用另一个容器已有的服务。

4.1 容器网络基础

作为一个寄宿在宿主主机上的容器，我们要想办法让外部网络能够访问到它，这样才能够使用其提供的服务。当Docker启动时，它会在宿主主机上创建一个名为docker0的虚拟网络接口。通过ifconfig命令，可以看到本机的网络接口情况：

```
$ ifconfig
docker0   Link encap:以太网  硬件地址 56:84:7a:fe:97:99
          inet 地址:172.17.42.1  广播:0.0.0.0  掩码:255.255.0.0
          inet6 地址: fe80::5484:7aff:fefe:9799/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:114 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:72 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:8314 (8.3 KB)  发送字节:8673 (8.6 KB)
...
eth1      Link encap:以太网  硬件地址 b8:ee:65:d5:5a:71
          inet 地址:192.168.222.224  广播:192.168.222.255  掩码:255.255.255.0
          inet6 地址: fe80::baee:65ff:fed5:5a71/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:1448650 错误:0 丢弃:0 过载:0 帧数:13734576
          发送数据包:645668 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:797901530 (797.9 MB)  发送字节:91342786 (91.3 MB)
          中断:17
...
```

可以看到，在宿主主机上有一个名为docker0的网络接口，其地址是172.17.42.1，这是Docker

默认分配的地址。关于Docker网络更多高级的主题，我们会在第5章中讲解。接下来，我们概要介绍一些数据管理涉及的网络基础，例如暴露网络端口和查看网络配置等。

4.1.1　暴露网络端口

当在Docker中运行网络应用时，我们需要在外部访问Docker中运行的应用，这时需要通过-P或者-p参数来指定端口映射。通过端口映射来实现端口暴露是容器对外提供服务的基础方法。

- -P（大写）参数。使用-P参数，Docker会在宿主主机上随机为应用分配一个49000～49900内的未被使用的端口，并将其映射到容器开放的网络端口。

  接下来，举一个这方面的例子，这里需要用到Docker官方提供的一个培训示例镜像training/webapp。由于本地并没有这个镜像，所以需要去搜索和下载。首先，通过如下命令来搜索：

```
$ docker search training/webapp
NAME                      DESCRIPTION   STARS     OFFICIAL   AUTOMATED
training/webapp                         8                    [OK]
amouat/webapp-training                  0                    [OK]
```

  可以看到，第一个是我们想要的镜像。接下来，我们需要运行这个镜像，Docker会自动为我们下载它：

```
$ docker run -d -P training/webapp python app.py
micall@micall-ThinkPad:~/docker$ docker run -d -P training/webapp python app.py
Unable to find image 'training/webapp' locally
Pulling repository training/webapp
31fa814ba25a: Download complete
511136ea3c5a: Download complete
...
Status: Downloaded newer image for training/webapp:latest
45fd5b0fc80413484da26bb68640b8794dc5409d0ff1bcab8a60c5541205592a
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
45fd5b0fc804        training/webapp:latest    "python app.py"     8 minutes ago
Up 8 minutes        0.0.0.0:49153->5000/tcp   condescending_kirch
```

  这时以training/webapp镜像为基础的容器已经在后台运行起来了，Docker为它起了一个随机的名字condescending_kirch。这里注意一下PORTS列，可以看到其值为0.0.0.0:491543->5000/tcp，Docker将宿主主机的49153端口映射到了容器的5000端口，并且遵守的协议是TCP协议。

  有了这个端口映射，我们就可以通过Web浏览器来访问，如图4-1所示。

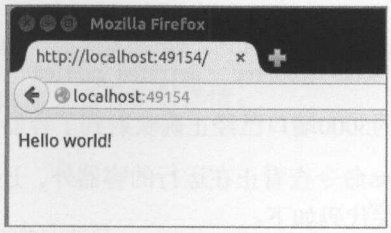


图4-1 Web应用测试主页

可以看到，这里可以正常显示“Hello world!”，这说明这个Web应用运行正常。我们可以用docker logs命令来查看日志输出，相关代码如下：

```
$ docker logs -f suspicious_stallman
 * Running on http://0.0.0.0:5000/
172.17.42.1 - - [16/Dec/2014 09:01:10] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [16/Dec/2014 09:01:10] "GET /favicon.ico HTTP/1.1" 404 -
172.17.42.1 - - [16/Dec/2014 09:02:17] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [16/Dec/2014 09:02:17] "GET /favicon.ico HTTP/1.1" 404 -
172.17.42.1 - - [16/Dec/2014 09:02:17] "GET /favicon.ico HTTP/1.1" 404 -
```

可以看出，我们对该网址进行了两次访问。

- **-p（小写）参数**。它可以指定宿主主机上的端口映射到容器内部指定的开放端口，格式有如下3种：

```
ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort
```

-p（小写）的情况比-P（大写）要复杂，下面我们分情况来说明一下。

  - hostPort:containerPort：当使用这种格式时，宿主主机中的所有网络接口都会被绑定。例如，我们想把宿主主机中所有网络接口的80端口映射到容器的5000端口：

```
$ docker run -d -p 80:5000 training/webapp python app.py
c56833c127474cac15fe7ce111057c35f32b5023b623ae3ea221d7b0f3a38af5
2014/12/16 17:24:31 Error response from daemon: Cannot start container
c56833c127474cac15fe7ce111057c35f32b5023b623ae3ea221d7b0f3a38af5: Error starting userland proxy:
    listen tcp 0.0.0.0:80: bind: address already in use
```

可以发现，绑定失败！仔细看出错信息，是绑定失败，80端口已经被使用。通过netstat命令，我们发现该端口被宿主主机的Apache程序占用了。我们把绑定换成和容器内部端口一样的5000，重新指定一遍，操作如下：

```
$ docker run -d -p 5000:5000 training/webapp python app.py
791116a0598ec3a90c8a24ecd2264c4455cd92f8ce3a944f815ddb0ba04eefbf
```

结果返回成功，我们可以通过docker ps命令查看一下具体情况：

```
$ docker ps
CONTAINER ID    IMAGE                    COMMAND             CREATED
```

```
STATUS          PORTS                    NAMES
791116a0598e    training/webapp:latest   "python app.py"     22 seconds ago
Up 22 seconds   0.0.0.0:5000->5000/tcp   agitated_galileo
```

可以看到，宿主主机的5000端口已经正确映射到了容器的5000开放端口。

除了可以使用docker ps命令查看正在运行的容器外，还可以使用 docker port命令查看一个容器的端口，相关代码如下：

```
$ docker port agitated_galileo  5000
0.0.0.0:5000
```

可以看到，容器的5000端口被宿主主机的5000映射。

- ip:hostPort:containerPort：指定IP地址的指定端口和容器的指定开放端口映射。例如，我们可以将环回地址上的5000端口映射到容器的5000端口，具体如下：

```
$ docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
3dd912f95d222d9f9a0331db688c9ed2b58b450637851d32baa43a1908271bd9
```

这里成功返回容器ID，说明执行正确。通过docker ps命令，我们可以查看具体情况：

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
3dd912f95d22        training/webapp:latest    "python app.py"     30 seconds ago
Up 29 seconds       127.0.0.1:5000->5000/tcp  backstabbing_turing
```

这里我们将宿主主机的环回地址上的5000端口和容器的5000端口进行映射。

- ip::containerPort：指定IP的随机端口映射到容器的指定端口。例如，我们想把容器内的5000端口和环回地址上的随机端口进行映射，可以这么做：

```
$ docker run -d -p 127.0.0.1::5000 training/webapp python app.py
8ea0edbe514c0a7a65cdeef1f7b66113d4aa52a2bb3a3a19dade1b320a236e29
```

我们发现映射成功。通过docker ps命令查看详情：

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
8ea0edbe514c        training/webapp:latest    "python app.py"     16 seconds ago
Up 16 seconds       127.0.0.1:49153->5000/tcp drunk_ptolemy
```

可以发现，Docker为我们随机分配了环回地址上的49153端口和容器的5000端口进行绑定。我们可以将环回地址替换成宿主主机上的其他IP地址。

当容器暴露多个端口的时候，我们可以对每个端口一一进行映射。

例如，我们想将宿主主机上的5000端口和容器内的5000端口映射，4000端口和容器内的80端口映射，具体为：

```
$ docker run -d -p 5000:5000 -p 4000:80 training/webapp python app.py
cf9c2732c68ee862faa5ba6c237c48491312b96448cbabcadf67a979da2d07dc
```

成功返回后，使用docker ps命令查看映射详情：

```
docker ps
CONTAINER ID    IMAGE                                COMMAND             CREATED
STATUS          PORTS                                NAMES
cf9c2732c68e    training/webapp:latest               "python app.py"     11 seconds ago
Up 10 seconds   0.0.0.0:4000->80/tcp, 0.0.0.0:5000->5000/tcp  hopeful_kirch
```

可以看到，4000端口和80端口映射，5000端口和5000端口映射。

4.1.2 查看网络配置

在第2章中，我们介绍过，通过docker inspect命令可以查看容器的配置信息。这里我们可以根据格式化过滤查看容器中网络相关的配置，相关代码如下：

```
$ docker inspect  --format '{{.NetworkSettings}}' cf9c27
map[PortMapping:<nil> Ports:map[5000/tcp:[map[HostIp:0.0.0.0 HostPort:5000]]
80/tcp:[map[HostIp:0.0.0.0 HostPort:4000]]] Bridge:docker0 Gateway:172.17.42.1 IPAddress:172.17.0.12
    IPPrefixLen:16 MacAddress:02:42:ac:11:00:0c]
```

可以看到端口映射、网桥、网关、IP以及物理地址等信息。

此外，也可以直接使用docker inspect命令找到相应的字段，操作为：

```
$ docker inspect cf9c27
```

得到的输出如下：

```
"NetworkSettings": {
    "Bridge": "docker0",
    "Gateway": "172.17.42.1",
    "IPAddress": "172.17.0.12",
    "IPPrefixLen": 16,
    "MacAddress": "02:42:ac:11:00:0c",
    "PortMapping": null,
    "Ports": {
        "5000/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": "5000"
            }
        ],
        "80/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": "4000"
            }
        ]
    }
},
```

这种JSON形式的数据展示层次相对清晰，更为易读。但直接使用docker inspect命令就意味着所有信息都将列出，所以两种方式各有所长和不足。

如果你仅仅需要查看该容器的IP地址，可以这样：

```
$ docker inspect  --format '{{.NetworkSettings.IPAddress}}' cf9c27
172.17.0.12
```

我们可以在宿主主机ping该地址：

```
$ ping 172.17.0.12
PING 172.17.0.12 (172.17.0.12) 56(84) bytes of data.
64 bytes from 172.17.0.12: icmp_seq=1 ttl=64 time=0.069 ms
64 bytes from 172.17.0.12: icmp_seq=2 ttl=64 time=0.039 ms
64 bytes from 172.17.0.12: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 172.17.0.12: icmp_seq=4 ttl=64 time=0.043 ms
^C
--- 172.17.0.12 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.039/0.050/0.069/0.012 ms
```

网络可达。

4.2　数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，使用它可以达到如下目的。

- 绕过“拷贝写”系统，以达到本地磁盘I/O性能。
- 绕过“拷贝写”系统，有些文件不需要在docker commit的时候打包进镜像中。
- 在多个容器之间共享目录。
- 在宿主和容器之间共享目录。
- 在宿主和容器之间共享单个文件（可以是socket）。

在这一节中，我们将讲解如何创建一个数据卷、共享一个数据卷以及在多个容器之间共享宿主主机目录。

4.2.1　创建数据卷

可以通过两种方式来创建数据卷，具体如下所示。

- 在Dockerfile中，使用VOLUME指令，如：

```
...
VOLUME /var/lib/postgresql
...
```

- 在命令行中使用docker run时，使用-v参数来创建数据卷并将其挂载到容器中，具体操作为：

```
$ docker run -d -P -v /webapp training/webapp python app.py
935934e00e9d96ae7ce3e0db311466e4c706d137e28f6b34147978d6777198b0
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
935934e00e9d        training/webapp:latest    "python app.py"     5 seconds ago
Up 3 seconds        0.0.0.0:49153->5000/tcp   suspicious_hoover
```

可以看到，这里定义了/webapp数据卷。

此外，我们可以通过docker inspect命令查看容器的数据卷，定位到数据卷相关的字段内容，具体为：

```
$ docker inspect suspicious_hoover
...
"Volumes": {
    "/webapp":
        "/var/lib/docker/vfs/dir/6fd914759fac2a0bff479e94a5961ec8b3a8505fc7f95e4921d24ff9f9d3ab74"
},
"VolumesRW": {
    "/webapp": true
}
...
```

当然，也可以直接通过格式化参数--format来查看指定数据卷部分的数据，具体操作为：

```
$  docker inspect --format {{.Volums}} suspicious_hoover
map[/webapp:/var/lib/docker/vfs/dir/6fd914759fac2a0bff479e94a5961ec8b3a8505fc7f95e4
921d24ff9f9    d3ab74]
```

可以看出，默认情况如果只是声明数据卷而没有映射到宿主主机上的具体目录，Docker会在/var/lib/docker/vfs/dir/下分配一个具有唯一名字的目录给该数据卷。

我们可以通过在宿主主机上使用ls参数验证该目录是否存在：

```
$ sudo ls -l /var/lib/docker/vfs/dir/
总用量 4
drwxr-xr-x 2 root root 4096 12月 17 14:11
6fd914759fac2a0bff479e94a5961ec8b3a8505fc7f95e4921d24ff9f9d3ab74
```

可以看到，在宿主主机上已经为该数据卷建立了一个独一无二的目录。

4.2.2 挂载主机目录作为数据卷

除了上述的仅仅声明一个数据卷外，我们还可以指定宿主主机上的某个目录作为数据卷。例如，我想把当前目录挂载为容器的/opt/webapp数据卷，具体操作为：

```
$ docker run -d -P --name webapp -v .:/webapp training/webapp python app.py
41be1170a753614d96aa2d7f0eb05d9b16b174be84ae673312a0a984cae009c6
2014/12/17 14:44:24 Error response from daemon: Cannot start container
41be1170a753614d96aa2d7f0eb05d9b16b174be84ae673312a0a984cae009c6: cannot bind mount volume: . volume
    paths must be absolute.
```

提示挂载出错！为什么呢？输出提示为数据卷的路径必须是绝对路径。所以“.”这个相对路径是会挂载失败的。正确的做法是：

```
$ docker run -d -P --name webapp -v `pwd`:/webapp training/webapp python app.py
295d19d19fbb0880eab75aa0e154be7f17cb1bb47bf2a67b4e5ec977470a91dd
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
295d19d19fbb        training/webapp:latest    "python app.py"     2 seconds ago
Up 2 seconds        0.0.0.0:49164->5000/tcp   webapp
```

查看并定位到数据卷字段信息：

```
$ docker inspect webapp
...
"Volumes": {
   "/webapp": "/home/micall/docker"
},
"VolumesRW": {
  "/webapp": true
}
...
```

可以看到，在Volumes字段中，容器内的/webapp目录映射到了宿主主机上的/home/micall/docker目录。

通过挂载本地目录到容器的数据卷，宿主主机对目录内的改变会同步反映到容器中，反过来也一样。我们可以利用这种方法把本地的一些数据复制到挂载目录下，提供给容器处理。容器也可以将处理结果写到该目录下，方便本地用户查看。需要说明的是，如果容器内部已经存在/webapp目录，那么挂载宿主目录之后，它的内容将会被覆盖。

需要注意的是，Dockerfile并不支持挂载本地目录到数据卷，这主要是因为不同操作系统的目录格式不尽相同。为了保证Dockerfile的移植性，所以不支持挂载本地目录到数据卷。

在上述inspect输出信息中，有一个字段是VolumesRW，配置的是数据卷的读写权限，默认情况下是读写，我们可以改变它的读写权限。具体做法是在创建数据卷的后面跟上权限控制，例如要将webapp数据卷由默认的wr可读可写变成ro只读。具体操作为：

```
$ docker run -d -P --name webapp -v `pwd`:/webapp:ro training/webapp python app.py
$ docker inspect webapp
...
"Volumes": {
    "/webapp": "/home/micall/docker"
},
"VolumesRW": {
    "/webapp": false
}
...
```

可以看到，VolumesRW中/webapp的值为false。

4.2.3 挂载主机文件作为数据卷

除了可以将主机目录挂载为数据卷外，还可以将单个主机的文件挂载为容器的数据卷。例如，在本地的当前目录下建立一个text.txt文件，里面输入一行“hello world”，然后将该文件挂载为数据卷。具体操作如下：

```
$ touch test.txt
$ gedit test.txt
$ docker run --rm -it -v ~/docker/test.txt:/test.txt ubuntu:latest /bin/bash
root@6e3ece82bfbc:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  test.txt  usr
boot  etc  lib   media  opt  root  sbin  sys  tmp       var
root@6e3ece82bfbc:/# cat test.txt
hello world
root@08554d5bc816:/# vi test.txt
root@08554d5bc816:/# cat test.txt
hello world
hello docker
root@08554d5bc816:/# exit
exit
micall@micall-ThinkPad:~/docker$ cat test.txt
hello world
hello docker
```

在上述操作中，我们先在本地文件系统中创建了一个test.txt，并在里面添加了一行“hello world”，然后运行ubuntu容器，并将刚刚新建的text.txt文件挂载到容器内的/text.txt。ls命令的输出表明该文件映射成功了，使用cat命令查看该文件的内容，输出即为hello world。然后在容器中通过vi编辑器编辑该文件，加入hello docker这行字符串。退出容器，到本地文件系统打开该文件，发现容器对它的改变也会同步到本地。

4.2.4 数据卷容器

数据卷容器是指一个专门用于挂载数据卷的容器，以供其他容器引用和使用。它主要用在多个容器需要从一处获得数据时。在实际操作时，需要将数据容器命名，有了确定的容器名之后，对它有依赖关系的其他容器就可以通过--volumes-from引用它的数据卷。

首先，建立一个数据卷容器，名为dbdata,并为该容器新建数据卷/dbdata。具体操作为：

```
$ docker run -d -v /dbdata --name dbdata training/postgres
a8d875ccbfbbf9188b450fc9aee68f34581d6f6cd39306a196991da73feaf394
$ docker ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED
STATUS              PORTS                       NAMES
a8d875ccbfbb        training/postgres:latest    "su postgres -c '/us   3 seconds ago
Up 2 seconds        5432/tcp                    dbdata
```

接着创建一个容器db1，它引用dbdata的数据卷，具体操作为：

```
$ docker run -d --volumes-from=dbdata --name db1 training/postgres
24ea52056abb9f2b2f02f22c421f2e901072a28de437e9442e01bdcf85906614
```

为了验证db1引用的是dbdata的数据卷，可以通过docker inspect命令来分别查看db1和dbdata容器，具体操作为：

```
$docker inspect dbdata
...
"Volumes": {
    "/dbdata":
       "/var/lib/docker/vfs/dir/886e6809b26d7dd765f1f59e93ff5be93d8c30a91e9e5ab14a244e638b
       d946e1"
},
"VolumesRW": {
    "/dbdata": true
}
...
$docker inspect db1
...
"Volumes": {
    "/dbdata":
       "/var/lib/docker/vfs/dir/886e6809b26d7dd765f1f59e93ff5be93d8c30a91e9e5ab14a244e638b
       d946e1"
},
"VolumesRW": {
    "/dbdata": true
}

...
```

可以看到，二者的数据卷是一样的。也就是说，db1容器和dbdata容器使用的是同一个数据卷/dbdata。需要说明的是，数据卷一旦声明，它的生命周期和声明它的那个容器就无关了。当声明它的容器停止了，数据卷也依然存在，除非所有引用它的容器都被删除了并且显式地删除了该数据卷。此外，一个容器引用一个数据卷容器时，并不要求数据卷容器是运行的。

我们可以让多个容器引用数据卷容器。例如，新建一个容器db2，它也引用dbdata容器的数据卷，相关操作为：

```
$ docker run -d --name db2 --volumes-from=db datatraining/postgres
```

此外，数据卷容器还可以级联引用。例如，新建一个容器db3，它引用db1容器的数据卷，具体操作为：

```
$ docker run -d --name db3 --volumes-from=db1 training/postgres
```

同样，可以用docker inspect查看其数据卷，结果是一样的。和dbdata、db1、db2一样，它们共用一个数据卷。

无论是声明数据卷的容器还是后续引用该数据卷的容器，容器的停止和删除都不会导致数据卷本身删除。如果需要删除数据卷，那么需要删除所有依赖它的容器，并且在删除最后一个依赖

容器时加入-v标志。这里，假如dbdata、db1和db2都已经删除了，那么删除db3的时候加上-v参数，就可以删除数据卷，具体为：

```
$ docker rm -v db3
```

此时你会发现在/var/lib/docker/vfs/dir/目录下就没有该数据卷对应的目录了。

4.2.5 数据的备份与恢复

利用数据卷容器，还可以进行数据的备份和恢复等。

1. 备份

利用数据卷容器，我们可以备份一个数据卷容器的数据。

首先，建立了一个数据卷容器，相关操作为：

```
$ docker run -d -v /dbdata --name dbdata training/postgres
```

这里通过-v创建了数据卷/dbdata，并将容器命名为dbdata。假如容器在运行过程中把产生的数据都保存到了/dbdata目录下。现在，我们想把它的数据备份到本地，相关操作为：

```
$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf
/backup/backup.tar /dbdata
```

这里我们通过ubuntu镜像新建了一个容器，它引用了dbdata容器的数据卷，并创建了一个数据卷映射，将本地文件系统的当前工作目录映射到容器的/backup目录。通过tar命令将/dbdata目录打包到/backup数据卷中，而该数据卷又因映射到了本地，所以dbdata容器的数据卷内的数据就保存到了本地的当前目录，文件名为backup.tar。

2. 恢复数据

恢复数据和备份数据一样简单。我们的目的是将本地的备份压缩包解压并加载进某个容器内，让其基于该备份的数据继续运行。

首先，我们先声明一个需要恢复的数据容器，操作为：

```
$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

该命令通过ubuntu镜像创建名为dbdata2的容器，该容器还创建了数据卷/dbdata。

然后，我们利用另一个引用它的容器来关联到本地目录，并将本地的数据解压进数据卷中去，具体做法为：

```
$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf
/backup/backup.tar
```

该命令使用busybox镜像创建一个容器，该容器引用dbdata2的数据卷，并且也创建了另一个

数据卷/backup，/backup被映射到本地文件系统中的当前目录。本地的当前目录存有backup.tar压缩包，然后通过tar解压到/dbdata数据卷下，这样就把备份好的数据重新恢复到容器内部。

4.3　容器连接

在4.1节中，我们使用了-P或者-p来暴露容器端口，以供外界使用该容器。在这一节中，我们要说明另一种容器对外提供服务的方法——容器连接。容器连接包含源容器和目标容器：源容器是提供服务的一方，对外提供指定服务；目标容器连接到源容器后，就可以使用其所提供的服务。容器连接依赖于容器名，所以当需要使用容器连接时，首先需要命名容器，然后使用--link参数进行连接。

4.3.1　容器命名

容器连接依赖于容器的名字。虽然容器启动后，Docker会自动为容器赋予一个名字，但是这个随即分配的名字并没有实质的意义。自己给容器命名有如下两个好处。

- 一个有意义的名字能够表明该容器的用途，这样方便记忆。例如，你可以将一个Web应用容器命名为web，而Docker自动分配名字的话，或许它就是david。很明显，我们更喜欢前者。
- 命名后的容器在容器连接中能够清晰地表征容器之间的逻辑依赖关系。例如，一个源容器里面包含的是数据库应用，我们将其命名为dbdata容器，然后目标容器是一个Web应用容器，这样连接的时候，我们就知道Web应用需要用到数据库服务。

介绍完容器连接中容器命名的重要性之后，现在通过--name参数来命名容器，具体操作为：

```
$ docker run -d -P --name web training/webapp python app.py
```

这里我们使用training/webapp镜像创建了一个名为web的容器，容器运行python命令。通过docker ps命令查看容器状况，如下：

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                     NAMES
d15c6e42de4b        training/webapp:latest    "python app.py"     4 seconds ago
Up 2 seconds        0.0.0.0:49166->5000/tcp   web
```

可以看到，我们的webapp容器已经已经运行起来了。

4.3.2　容器连接

连接的格式为--link name:alias，其中name是源容器的名称，alias是这个连接的别名。

接下来，我们通过一个Web应用来说明容器连接。一个web容器包含Web应用，它需要使用另外一个dbdata容器的数据库服务，它们之间采用连接互联。

首先，建立一个数据库容器dbdata，相关操作为：

```
$ docker run -d --name dbdata training/postgres
```

然后，建立一个Web容器web，将其连接到dbdata容器，具体操作为：

```
$ docker run -d -P --name web --link dbdata:db training/webapp python app.py
```

这条命令以training/webapp镜像来创建名为web的容器，容器通过--link链接dbdata，连接的别名为db。-P参数表明端口映射是随机进行的。通过该条命令，web容器和dbdata容器就成功建立了连接。

接着，通过docker inpsect命令，看到和连接相关的字段，具体如下：

```
$docker inspect web
...
"Links": [
    "/dbdata:/web/db"
],
...
```

这里记录了本容器的连接关系。

通过这种方式，dbdata容器为web容器提供了服务，但并没有像-P（或者-p）参数那样，让容器对外暴露端口，这使得源容器dbdata更安全。既然web容器和dbdata容器之间已经建立了连接，那么web是如何使用dbdata的服务的呢？

Docker给目标容器提供了如下两种方式来暴露连接提供的服务：

- 环境变量；
- /etc/hosts文件。

下面我们分别说明它们。

1. 环境变量

当两个容器通过连接互联之后，Docker将会在目标容器中设置相关的环境变量，以便在目标容器中使用源容器提供的服务。连接环境变量的的命名格式为<alias>_NAME，其中alias是--link参数中的别名。例如web容器连接dbdata容器，参数为--link dbdata:webdb，那么在web容器中则有环境变量WEBDB_NAME=/web/webdb。

一般情况下，可以使用env命令来查看一个容器的环境变量，相关代码为：

```
$ docker run --rm --name web2 --link dbdata:webdb training/webapp env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=ae63f23dc769
```

```
WEBDB_PORT=tcp://172.17.0.32:5432
WEBDB_PORT_5432_TCP=tcp://172.17.0.32:5432
WEBDB_PORT_5432_TCP_ADDR=172.17.0.32
WEBDB_PORT_5432_TCP_PORT=5432
WEBDB_PORT_5432_TCP_PROTO=tcp
WEBDB_NAME=/web2/webdb
WEBDB_ENV_PG_VERSION=9.3
HOME=/
```

可以看到，和连接相关的前缀都是WEBDB。除了刚刚说到的WEBDB_NAME，还有很多PORT相关的环境变量——<name>_PORT_<port>_<protocol>，其中<name>是--link的别名（webdb），port是暴露的端口，<protocol>是TCP或者UDP协议。例如，上面案例中的WEBDB_PORT_5432_TCP=tcp://172.17.0.32:5432，该等式的右边是一个URL，其格式为<protocol>://<contianer_ip_address>:<port>（例如tcp://172.17.0.32:5432），而这个URL会分成三部分。

- <name>_PORT_<port>_<protocol>_ADDR：地址。
- <name>_PORT_<port>_<protocol>_PORT：端口。
- <name>_PORT_<port>_<protocol>_PROTO：协议。

2. /etc/hosts文件

查看目标容器的/etc/hosts配置文件，具体操作如下：

```
$ docker run -i -t --rm --name web2 --link dbdata:webdb training/webapp /bin/bash
root@ad5dc3c7378d:/opt/webapp# cat /etc/hosts
172.17.0.8 ad5dc3c7378d
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.32 webdb
```

可以看到，容器连接webdb对应的地址为172.17.0.32，该地址实为dbdata容器的地址，容器对webdb连接的操作将会映射到该地址上。

4.3.3　代理连接

在上一节中，我们讲到的容器连接都是在一个宿主主机上的连接。就目前而言，对于跨主机的容器连接，Docker并没有给出有效的方法。在Docker的官方网站上，我们看到利用ambassador模式可以实现跨主机连接，我们称这种模式的连接叫作代理连接。

通过代理连接，可以解耦两个原本直接相连的容器的耦合性。下面看一个例子，如图4-2所示，redis-client是客户端容器，它需要使用redis-server容器提供的服务，它们之间采用直接相连的方式进行连接。

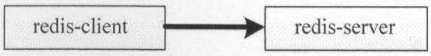


图4-2 redis-client直接依赖redis-server

这种直接相连存在两个问题。

- 不能够跨主机连接。
- 耦合性太高。假如redis-client需要连接到新的redis-server，那么必须先重启redis-client容器本身。容器重启意味着容器内部所有应用和服务的中断，这在实际产品环境中有时候是成本太高，有时候根本不允许。所以，我们需要一种方法来解耦二者的关系，使得redis-client无需关心它连接到的是哪一个redis-server。

通过代理连接，可以实现如图4-3所示的连接。客户端主机上的redis-client容器连接到同一主机的ambassador 1代理容器，ambassador 1容器通过网络连接到服务器主机上的ambassador 2代理容器，ambassador 2容器连接到redis-server容器，最终实现redis-client容器使用主机2上redis-server提供的redis服务。

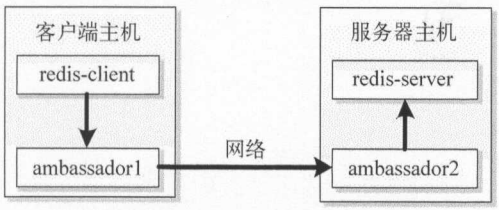


图4-3 通过两个代理进行连接

接下来，我们一步步说明如何建立上述代理连接。

(1) 在服务器主机上启动一个装有redis服务的容器，具体操作为：

```
server$ sudo docker run -d --name redis crosbymichael/redis
```

(2) 在服务器主机上建立一个代理容器ambassador 2，将它连接到redis-server，具体操作为：

```
sudo docker run -d --link redis:redis --name ambassador2 -p 6379:6379 ambassador
```

(3) 客户端主机上包含容器redis-client，它需要使用redis-server容器中的redis服务。我们需要先建立一个代理容器ambassador 1，将它连接到服务器主机的代理容器ambassador 2。具体如下：

```
client$ sudo docker run -d --name redis_ambassador --expose 6379 -e
REDIS_PORT_6379_TCP=tcp://192.168.1.52:6379 svendowideit/ambassador
```

(4) 在客户端主机上如果需要使用redis服务，则只需要连接到本机的redis_ambassador容器，具体操作为：

```
client-server $ sudo docker run -i -t --rm --link redis_ambassador:redis
```

```
relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

这样就可以直接使用redis服务了。

总结上面的步骤，通过代理连接，我们就完成了一次跨主机的容器连接。

Part 2 第二篇

案例篇：综合案例

本篇内容

第5章 创建SSH服务镜像

在第2章中，我们介绍了在容器内部管理容器的命令，例如attach、exec等，但是这些命令无法解决远程管理容器的需求。在现实环境中，服务器都在远端的机房，Linux系统管理员需要通过SSH服务连接到远端系统进而管理系统。Docker的很多系统镜像并没有安装SSH服务，这就要求我们自己为其安装SSH服务。

本章将具体介绍如何通过如下两种方式来构建带SSH服务的系统容器：

- 基于commit命令的方式；
- 基于Dockerfile的方式。

5.1 基于 commit 命令的方式

Docker的commit命令提供了将用户修改过的容器提交成为新镜像的功能。现在，我们就使用docker commit命令来生成支持SSH服务的镜像。

1. 准备工作

我们的目标镜像是以ubuntu:14.04为基础的，首先使用ubuntu:14.04镜像创建一个新的容器：

```
$ sudo docker run -it ubuntu /bin/bash
```

执行该命令后，我们已经进入容器的命令终端。

接下来，配置容器内系统的软件源。由于系统默认的软件源服务器都在国外，下载速度可能会很慢，所以可以使用国内的源，常用的有163、sohu等，教育网的用户可以使用电子科技大学、北京理工大学的源，这里我们使用的是163的源。其他版本的系统或其他的源可以自行到网上搜索。将下面的内容追加到/etc/apt/sources.list文件后面：

```
deb http://mirrors.163.com/ubuntu/ precise main universe restricted multiverse
deb-src http://mirrors.163.com/ubuntu/ precise main universe restricted multiverse
deb http://mirrors.163.com/ubuntu/ precise-security universe main multiverse restricted
deb-src http://mirrors.163.com/ubuntu/ precise-security universe main multiverse restricted
deb http://mirrors.163.com/ubuntu/ precise-updates universe main multiverse restricted
deb http://mirrors.163.com/ubuntu/ precise-proposed universe main multiverse restricted
```

```
deb-src http://mirrors.163.com/ubuntu/ precise-proposed universe main multiverse restricted
deb http://mirrors.163.com/ubuntu/ precise-backports universe main multiverse restricted
deb-src http://mirrors.163.com/ubuntu/ precise-backports universe main multiverse restricted
deb-src http://mirrors.163.com/ubuntu/ precise-updates universe main multiverse restricted
```

然后执行sudo apt-get update命令，更新安装源。

2. 安装和配置SSH服务

更新完安装源后，就可以安装SSH服务了。使用apt-get安装openssh-server：

```
root@9598cb8e8f4a:/# apt-get install openssh-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
    ca-certificates krb5-locales libck-connector0 libedit2 libgssapi-krb5-2
    libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0
    libpython-stdlib libpython2.7-minimal libpython2.7-stdlib libwrap0 libx11-6
    libx11-data libxau6 libxcb1 libxdmcp6 libxext6 libxmuu1 ncurses-term
...
Processing triggers for ca-certificates (20130906ubuntu2) ...
Updating certificates in /etc/ssl/certs... 164 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.
Processing triggers for ureadahead (0.100.0-16) ...
root@9598cb8e8f4a:/#
```

安装的输出有很多。可以看到，有许多依赖包要安装，但apt都将会自动为你安装好。

虽然SSH服务安装好了，但是现在还运行不了SSH服务，因为SSH需要的一个目录/var/run/sshd不存在。这里手动创建该目录：

```
root@9598cb8e8f4a:/var/run# mkdir -p /var/run/sshd
```

接着，以后台方式启动SSH服务：

```
root@9598cb8e8f4a:/var/run# /usr/sbin/sshd  -D &
[1] 3263
```

SSH默认监听22端口。为了验证服务是否成功启动，可以查询容器系统的端口状态：

```
root@9598cb8e8f4a:/var/run# netstat -natp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
```

可以看到，22号端口已经处于LISTEN状态了，说明SSH服务已经正常运行了。

默认情况下，pam认证程序会对我们的SSH登录进行限制。修改pam对SSH的配置，取消登录限制，具体操作为编辑/etc/pam.d/sshd，将下面这行注释掉（在前面加#）：

```
session    required     pam_loginuid.so
```

3. 生成和添加公钥

接下来，添加允许登录的用户的公钥。假如我们想允许B主机上的root用户可以通过SSH登录到容器内部，那么首先要在B主机上通过ssh-keygen -t rsa命令生成root用户的公钥。记住，下面的命令是在B主机上而不是容器系统上进行的：

```
root@B:# ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
36:e1:cf:79:c6:97:a3:f3:1c:97:cf:6a:c8:0d:fd:41 root@B
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|        .        |
|       . .     E |
|        S    ..  |
|       . + o. .o.|
|          +.++=oo|
|           o++o=o|
|            .++.o|
+-----------------+
root@B:~# cd .ssh
root@B:~/.ssh# ls
id_rsa  id_rsa.pub
```

~/.ssh目录下的id_rsa.pub文件即为该主机上root用户的公钥，接下来我们需要将其添加到容器系统内。

回到容器系统内部，在/root目录下创建.ssh目录，然后将允许远程登录到容器系统的用户的公钥添加到authorized_keys文件中：

```
root@9598cb8e8f4a:/var/run# cd ~
root@9598cb8e8f4a:~# mkdir .ssh
root@9598cb8e8f4a:~# vi .ssh/authorized_keys
```

4. 制作SSH运行脚本

至此，公钥添加完毕。接下来，我们需要创建启动SSH服务的脚本run.sh，并为其添加执行权限。由于容器启动时只能运行一个命令，所以一般把要启动的程序和服务都放在一个脚本中，这样只要运行这个脚本就可以了。目前，虽然只有SSH服务一个程序，我们还是用统一的脚本来处理。如下面的命令所示，第一行通过vi命令编辑启动脚本，第二行为刚才的脚本添加执行权限，第三行向我们展示了脚本的内容：

```
root@9598cb8e8f4a:~# vi run.sh
root@9598cb8e8f4a:~# chmod u+x run.sh
root@9598cb8e8f4a:~# cat run.sh
#!/bin/bash
/usr/sbin/sshd -D
```

需要注意的是，脚本中的命令不能添加&，否则后面生成的镜像会出现问题。比如，如果脚本是/usr/sbin/sshd -D &，则当容器以该脚本启动时，会立马执行完毕，相应的容器也就退出了，这显然不是我们想要的，我们要SSH服务一直监听着。此处的-D参数用于告诉SSH服务不以守护进程运行，而是和运行终端关联，有了关联终端，容器就不会退出。

最后，使用exit命令或者按ctrl+D组合键退出容器。

5. 提交生成镜像

使用docker commit命令将刚才的容器提交为一个新的镜像：

```
$ sudo docker commit 9598cb8e8f4a ssh:commit
9be5e20b8429fdf253ebe7826e7838f21468fcace7b8c3d61fbe5612efed79b0
```

使用docker images命令，可以查看我们新生成的镜像ssh: commit：

```
$ sudo docker images
REPOSITORY      TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ssh             commit              9be5e20b8429        26 seconds ago      230.3 MB
```

6. 使用镜像

使用上面的新镜像启动容器，并用-p参数添加端口映射2022:22。22是容器SSH服务监听的端口，2022是映射到主机的端口：

```
$ sudo docker run -d -p 2022:22 ssh:commit /root/run.sh
943315c382297af818a6d5b425303e541753913c587c928279bd355be9ecec43
```

容器启动成功后，可以使用docker ps命令查看我们的容器信息：

```
$ sudo docker ps
CONTAINER ID   IMAGE        COMMAND           CREATED              STATUS
PORTS   NAMES
943315c38229   ssh:commit  "/root/run.sh"   About a minute ago   Up 21 seconds
0.0.0.0:2022->22/tcp   romantic_mestorf
```

7. 通过SSH登录容器

在B主机的root用户下，通过访问容器的宿主主机的2022端口登录容器：

```
$ ssh 192.168.199.231 -p 2022
The authenticity of host '[192.168.199.231]:2022 ([192.168.199.231]:2022)' can't be established.
    ECDSA key fingerprint is f8:cc:7c:dd:bf:4e:d1:32:08:e3:11:0e:8c:0c:fd:e9.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.199.231]:2022' (ECDSA) to the list of known hosts.
...
root@943315c38229: ~#
```

成功通过SSH登录到容器系统。

5.2　基于 Dockerfile 的方式

在第3章中，我们已经介绍过Dockerfile的基础知识，下面我们将基于Dockerfile的方式来创建一个SSH服务的镜像。

1. 准备文件

创建一个存放生成镜像相关文件的工作目录：

```
$ mkdir ssh_dockerfile && cd ssh_dockerfile
```

在该目录下，需要创建3个文件：Dockerfile、run.sh和authorized_keys。Dockerfile用于构建镜像，run.sh是启动SSH服务的脚本文件，authorized_keys则是包含需要远程登录的用户的公钥，例如上一节中的B主机root用户的公钥。

run.sh脚本文件的内容如下：

```
#!/bin/bash
/usr/sbin/sshd -D
```

authorized_keys文件内容的生成方式和上一节一样：

```
$ ssh-keygen -t rsa
```

按回车选择默认的没有密码，此时会在当前用户目录下的.ssh目录中生成两个文件——id_rsa和id_rsa.pub，其中后者就是我们要的公钥文件。将其内容加到authorized_keys文件中，具体为：

```
$ cat ~/.ssh/id-rsa.pub > authorized_keys
```

2. 编写Dockerfile

Dockerfile用于创建镜像，下面列出了最终的Dockerfile：

```
#使用的基础镜像
FROM ubuntu:14.04
#添加作者信息
MAINTAINER kaixin 498849289@qq.com
#安装SSH服务
RUN apt-get install -y openssh-server
RUN mkdir -p /var/run/sshd
RUN mkdir -p /root/.ssh
#取消pam登录限制
RUN sed -ri 's/session    required     pam_loginuid.so/#session    required     pam_loginuid.so/g'
    /etc/pam.d/sshd
#添加认证文件和启动脚本
ADD authorized_keys /root/.ssh/authorized_keys
RUN echo "#!/bin/bash" > /root/run.sh
RUN echo "/usr/sbin/sshd -D" >> /root/run.sh
```

```
RUN chmod u+x /root/run.sh
#导出端口
EXPOSE 22
#设置默认的启动命令
CMD ["/root/run.sh"]
```

Dockerfile文件内部的RUN操作和上一节的操作基本相同，只是这里使用的是官方的源，没有添加源。

3. 创建镜像

运行docker build目录，生成我们的目标镜像：

```
$ sudo docker build ./
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> 8eaa4ff06b53
Step 1 : MAINTAINER kaixin 498849289@qq.com
 ---> Running in b2cfe2c1acf9
 ---> f40d6891324a
...
Step 11 : CMD /root/run.sh
 ---> Running in a8abe336cc23
 ---> 9c9743f2d33a
Removing intermediate container a8abe336cc23
Successfully built 9c9743f2d33a
```

看到最后的Successfully built，就表明镜像成功生成了，其中9c9743f2d33a是我们的镜像ID。

使用docker images命令查看本地镜像，可以看到我们生成的镜像，相关代码如下：

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ssh                 dockerfile          9c9743f2d33a        3 minutes ago       230.3 MB
ssh                 commit              9be5e20b8429        53 minutes ago      230.3 MB
```

其中ssh:dockerfile就是我们刚刚通过Dockerfile构建的镜像。现在启动它：

```
$ sudo docker run -d -p 2023:22 ssh:dockerfile /root/run.sh
```

然后新开启一个终端并将其连接到容器：

```
$ ssh 192.168.199.231 -p 2022
```

现在，我们通过两种方式实现了SSH，此时就可以通过远程机器来创建镜像或者对容器进行维护。我们可以以生成的SSH镜像为基础来构建新的镜像，如果使用Dockerfile的话，只要把SSH的Dockerfile加进去就可以了。

第6章 综合案例1：Apache+PHP+MySQL

本章是一个综合案例，这里将使用前面的知识，在基础镜像centos的基础上，搭建一个基于Apache、PHP和MySQL的Web应用。通过本章的案例，可以很好地巩固前面所学的基础知识，特别是Dockerfile的编写、多容器的应用等知识点。

图6-1是本案例容器的部署架构，其中包含两个容器：Web容器和数据库容器。Web容器运行着Apache和PHP服务，并包含PHP页面。数据库容器运行MySQL服务，保存应用的数据。Web容器需要使用数据库容器提供的服务，Web容器内的PHP页面将根据需求，访问数据库容器提供的数据，并将结果返回给Apache，最终展示给用户。

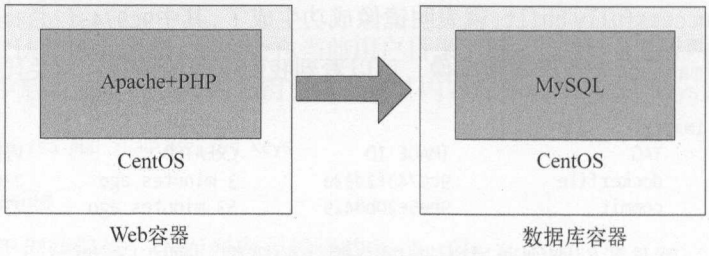


图6-1　Apache+PHP+MySQL容器架构

本章内容主要包含：

- 构建mysql镜像。通过Dockerfile来构建镜像，并将其上传到Docker Hub中。
- 构建apache+php镜像。
- 启动容器。

6.1　构建 mysql 镜像

MySQL是一款非常流行的开源关系型数据库，由于它具有快速、可靠和易于使用的特点，很多软件公司的数据都用它进行存储。MySQL能够运行在不同的系统平台之上，并且支持C、

C++、Eiffel、Java、Perl、PHP、Python、Ruby和Tcl等编程语言的API，其官网地址为http://www.mysql.com/。本节将介绍如何通过Dockerfile构建mysql镜像。

6.1.1 编写镜像 Dockerfile

本章的镜像都是基于基础镜像centos构建的，这里我将详细列出每一个步骤，并做出必要的解释。在实际应用中，我们也可以选择将Docker Hub上已有的容器作为开发的基础。

(1) 创建Dockerfile文件，并且在文件开始位置添加使用#注释的描述信息：

```
# 名称：容器化的MySQL
# 用途：用作后端数据持久化服务
# 创建时间：2015.01.20
```

这些注释的描述信息并不是必需的，但是我们推荐写上，这样方便其他使用者了解我们的容器以及后期的维护、更新。

(2) 新建的镜像必须要有一个镜像作为开发的基础。这里，定义我们使用的基础镜像：

```
FROM centos
```

(3) 声明维护者信息：

```
MAINTAINER kaixin 498849289@qq.com
```

(4)定义工作目录：

```
WORKDIR /root/
```

(5) 我们使用的CentOS是CentOS 7，默认并没有包含MySQL的源，所以要下载MySQL的源并安装它。先安装一个下载工具wget，然后再用wget来下载MySQL的源，最后将源安装到我们的系统上。由于在构建镜像的过程中没法交互，所以要在Dockerfile里使用-y参数，使每个需要确定的地方都选择Yes：

```
RUN yum -y install wget
RUN wget http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm
RUN rpm -ivh /root/mysql-community-release-el7-5.noarch.rpm
```

(6) 安装MySQL服务。这里使用yum安装MySQL：

```
RUN yum -y install mysql-server
```

安装好MySQL后，默认并没有建立数据库，需要使用mysql_install_db创建一个数据库：

```
RUN mysql_install_db --user=mysql
```

这里的数据库指的是数据库文件，MySQL保存所有数据的磁盘文件。后面的--user参数指定使用数据库的用户名，此处为mysql，该用户在MySQL服务安装完后在系统中被创建。从原理上

来说，也可以设置为--user=root，但一般不这么做，这是因为root用户的权限高于其他用户，会导致其他用户用不了MySQL服务。需要说明的是，此处的user用户指的是数据库物理文件的拥有者，而不是下面步骤中连接到数据库对数据库进行操作的用户，这点要区分清楚。

(7) 通过环境变量指定mysql使用的用户名和密码。MySQL拥有一个默认的用户root，但root用户默认只能在本地访问，所以这里定义了一个额外的用户test：

```
ENV MYSQL_USER test
ENV MYSQL_PASS mypassword
```

除了在Dockerfile中通过ENV来设置环境变量外，也可以在启动的时候通过-e参数进行设置。

(8) 让容器支持中文，centos容器默认是不支持中文的：

```
ENV LC_ALL en_US.UTF-8
```

(9) 建立数据库和表。目前，我们的数据库还是空的，里面没有数据，这里我用一个脚本来建立数据库和相关表：

```
ADD build_table.sh /root/build_table.sh
RUN chmod u+x /root/build_table.sh
RUN /root/build_table.sh
```

其中build_table.sh脚本的内容稍后给出。

(10) 导出3306端口（这是MySQL使用的端口），以使外部可以访问它：

```
EXPOSE 3306
```

(11) 定义默认的启动命令，这里使用一个脚本来启动mysql：

```
ADD run.sh /root/run.sh
RUN chmod u+x /root/run.sh
CMD /root/run.sh
```

其中run.sh的内容稍后给出。

以下是完整版的Dockerfile：

```
# 名称：容器化的MySQL
# 用途：用作后端数据持久化服务
# 创建时间：2015.01.20
FROM centos
MAINTAINER kaixin 498849289@qq.com
WORKDIR /root/
RUN yum -y install wget
RUN wget http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm
RUN rpm -ivh /root/mysql-community-release-el7-5.noarch.rpm
RUN yum -y install mysql-server
RUN mysql_install_db --user=mysql
ENV MYSQL_USER test
ENV MYSQL_PASS mypassword
```

```
#支持中文
ENV LC_ALL en_US.UTF-8
ADD build_table.sh /root/build_table.sh
RUN chmod u+x /root/build_table.sh
RUN /root/build_table.sh
EXPOSE 3306
ADD run.sh /root/run.sh
RUN chmod u+x /root/run.sh
CMD /root/run.sh
```

下面我们说明Dockerfile用到的两个脚本文件——build_table.sh和run.sh，它们和Dockerfile放在同一个目录下。

build_table.sh的内容为：

```
#!/bin/bash
mysqld_safe &
sleep 3
mysql -e "GRANT ALL PRIVILEGES ON *.* TO '$MYSQL_USER'@'%' IDENTIFIED BY '$MYSQL_PASS' WITH GRANT
    OPTION;"
mysql -e "create database scores"
mysql -e "create table scores.name_score(name char(20) not null,score int not null)DEFAULT CHARSET=utf8"
mysql -e "insert into scores.name_score values ('李明',80),('张军',90),('王小二',95)"
```

build_table.sh首先以后台方式运行mysqld_safe，然后通过mysql客户端程序执行数据库操作。第一条操作是授权给MYSQL_USER变量代表的用户，该变量的值我们在Dockerfile中设定为test了。第二条操作是创建scores数据库，第三条操作是创建name_score表格，它属于scores数据库，第四条则是向表中插入三条数据。

run.sh定义了容器的默认启动行为，这里只是拉起mysql，其内容为：

```
#!/bin/bash
mysqld_safe
```

6.1.2 构建和上传镜像

Dockerfile和必要的文件都准备好了，接下来就可以使用docker build命令来构建镜像了：

```
$ sudo docker build -t lqkaixin/centos-mysql:v1 ./
Sending build context to Docker daemon  7.68 kB
Sending build context to Docker daemon
Step 0 : FROM centos
 ---> 8efe422e6104
Step 1 : WORKDIR /root/
 ---> Using cache
 ---> 61b420a9c947
Step 2 : RUN yum -y install wget
 ---> Using cache
 ---> 655f7617edb4
....
```

```
Step 15 : RUN chmod u+x /root/run.sh
 ---> Using cache
 ---> 0b8e3b2ec3b1
Step 16 : CMD /root/run.sh
 ---> Using cache
 ---> 200f7c523709
Successfully built 200f7c523709
```

docker build命令的-t参数将构建成的镜像标记为lqkaixin/centos-mysql:v1。lqkaixin是命名空间，就是笔者在Docker Hub上的用户名，centos-mysql是仓库名，v1是标签，表明这是第一个版本。在输出的最后，我们看到Successfully built 200f7c523709，这表明镜像生成成功，其中200f7c523709就是镜像的ID。

构建好镜像之后，通过docker push命令将镜像提交到Docker Hub：

```
$ sudo docker push lqkaixin/centos-mysql:v1
The push refers to a repository [lqkaixin/centos-mysql:v1] (len: 1)
Sending image list
Please login prior to push:
Username: lqkaixin
Password:
Email: 498849289@qq.com
Login Succeeded
The push refers to a repository [lqkaixin/centos-mysql:v1] (len: 1)
Sending image list
Pushing repository lqkaixin/centos-mysql:v1 (1 tags)
511136ea3c5a: Image already pushed, skipping
5b12ef8fd570: Image already pushed, skipping
...
200f7c523709: Image successfully pushed
Pushing tag for rev [200f7c523709] on
{https://cdn-registry-1.docker.io/v1/repositories/lqkaixin/centos-mysql/tags/v1}
```

由于之前没有登录过，这里提示需要输入用户名、密码和注册邮箱。如果镜像比较大，上传会花费比较长的时间。看到类似最后一行输出，就表明上传成功。登录Docker Hub，就可以看到刚刚上传的镜像了。

现在mysql镜像就创建好了，并且可以在任何可以上网的机器上从Docker Hub拉取。

6.2　构建 apache+php 镜像

Apache是世界使用排名第一的Web服务器软件。由于其跨平台性和安全性而被广泛使用，它是目前最流行的Web服务器端软件。PHP是一种通用开源脚本语言，非常适用于Web开发，是常用的服务端脚本语言。在这一节里，我们将以centos镜像为基础，使用Dockerfile搭建一个apache+php镜像。

6.2.1 编写镜像 Dockerfile

(1) 和上一节类似，首先创建Dockerfile，在前面加入适当的说明：

```
# 名称：容器化的Apache+PHP
# 用途：用作Web前端服务
# 创建时间：2015.01.22
```

(2) 制定我们使用的基础镜像：

```
FROM centos
```

(3) 声明维护者信息：

```
MAINTAINER kaixin 498849289@qq.com
```

(4) 制定工作目录：

```
WORKDIR /root/
```

(5) 安装httpd和php。需要注意的是，安装过程中可能出现错误导致命令运行失败，进而中断build镜像。如果遇见错误但该错误又不影响应用的使用时，可以忽略该错误，以便安装顺利执行。我这里使用||true来保证整个命令返回true：

```
RUN yum -y install httpd php || true
```

(6) 安装MySQL客户端和php-mysqlnd。MySQL是用来和远程MySQL服务端通信的，而php-mysqlnd用于PHP和MySQL的沟通驱动：

```
RUN yum -y install mysql php-mysqlnd
```

(7) 创建必要的目录。要想Apache正常工作，就必须创建下面的目录：

```
RUN mkdir /var/log/httpd
RUN mkdir /var/www/
RUN mkdir /var/www/html/
```

其中，/var/log/httpd是存放日志的目录，var/www/是Apache应用的数据根目录，/var/www/html/是存放Web页面的目录。

(8) 通过环境变量定义远端MySQL的地址、用户名和密码。可以在启动容器时用-e改变：

```
ENV MYSQL_ADDR 172.17.0.36:3306
ENV MYSQL_USER test
ENV MYSQL_PASS mypassword
```

(9) 定义启动服务必需的环境变量TERM，并且定义支持中文：

```
ENV TERM linux
#支持中文
ENV LC_ALL en_US.UTF-8
```

(10) 添加我们的页面，这里我只定义了一个PHP页面。可以根据需要添加任意的页面或者挂载宿主主机已有的目录：

```
ADD test.php /var/www/html/test.php
```

ADD命令会将本地的test.php文件复制到容器内的/var/www/html/目录下。

(11) 导出Apache的服务端口，这里是默认的80端口：

```
EXPOSE 80
```

(12) 添加启动脚本，定义默认启动命令：

```
ADD run.sh /root/run.sh
RUN chmod u+x /root/run.sh
CMD /root/run.sh
```

下面是完整的Dockerfile：

```
# 名称：容器化的Apache+PHP
# 用途：用作Web前端服务
# 创建时间：2015.01.22
FROM centos
MAINTAINER kaixin 498849289@qq.com
WORKDIR /root/
RUN yum -y install httpd php || true
RUN yum -y install mysql php-mysqlnd
RUN mkdir /var/log/httpd
RUN mkdir /var/www/
RUN mkdir /var/www/html/
ENV MYSQL_ADDR 172.17.0.36:3306
ENV MYSQL_USER test
ENV MYSQL_PASS mypassword
ENV TERM linux
#支持中文
ENV LC_ALL en_US.UTF-8
ADD test.php /var/www/html/test.php
EXPOSE 80
ADD run.sh /root/run.sh
RUN chmod u+x /root/run.sh
CMD /root/run.sh
```

下面我们说明Dockerfile用到的两个文件test.php和run.sh。test.php是PHP演示页面文件，其内容为：

```
<?php
    $con = mysql_connect(getenv("MYSQL_ADDR"),getenv("MYSQL_USER"),getenv("MYSQL_PASS"));
    if (!$con)
    {
        die('失败: ' . mysql_error());
    }
    else
```

```
    {
        mysql_query("SET NAMES utf8");
        mysql_select_db("scores", $con);
        $result = mysql_query("SELECT * FROM name_score");
        while($row = mysql_fetch_array($result))
        {
            echo $row['name'] . " " . $row['score'];
            echo "<br />";
        }
    }
    mysql_close($con);
?>
```

test.php做的工作很简单，它连接到我们指定的MySQL服务端，查询scores.name_score表，并把查询结果按每条一行的方式输出。

run.sh用于启动Apache服务，其内容为：

```
#!/bin/bash
#启动Apache
httpd
#防止脚本结束
while true;do sleep 1000;done
```

httpd命令会立即返回，这里我们用了一个while循环防止脚本返回。

6.2.2 构建和上传镜像

编写好Dockerfile后，可以通过docker build命令构建镜像：

```
$ sudo docker build -t lqkaixin/centos-apache-php:v1 ./
Sending build context to Docker daemon  7.68 kB
Sending build context to Docker daemon
Step 0 : FROM centos
 ---> 8efe422e6104
Step 1 : WORKDIR /root/
 ---> Using cache
 ---> 61b420a9c947
Step 2 : RUN yum -y install httpd php || true
 ---> Using cache
 ---> 4a9af7deb20f
...
Step 15 : RUN chmod u+x /root/run.sh
 ---> Running in baa86fdc7849
 ---> f516e1aad85e
Removing intermediate container baa86fdc7849
Step 16 : CMD /root/run.sh
 ---> Running in c6e88bbbbf0f
 ---> 1d07b342055c
Removing intermediate container c6e88bbbbf0f
Successfully built 1d07b342055c
```

在docker build中，我们指定要生成的镜像的命名空间/仓库/标签为lqkaixin/centos-apache-php:v1。

构建好镜像之后，通过docker push命令提交镜像到Docker Hub：

```
$ sudo docker push lqkaixin/centos-apache-php:v1
The push refers to a repository [lqkaixin/centos-apache-php:v1] (len: 1)
Sending image list
Pushing repository lqkaixin/centos-apache-php:v1 (1 tags)
511136ea3c5a: Image already pushed, skipping
5b12ef8fd570: Image already pushed, skipping
...
1d07b342055c: Image successfully pushed
Pushing tag for rev [1d07b342055c] on
{https://cdn-registry-1.docker.io/v1/repositories/lqkaixin/centos-apache-php/tags/v1}
```

由于前面创建mysql镜像时已登录过，这里就不需要再次输入用户登录信息了。

6.3　启动容器

至此，我们已经有了mysql和apache容器，接下来启动并使用它们。

启动mysql容器：

```
$ sudo docker run --name test_mysql -d -P lqkaixin/centos-mysql:v1
643c9b48e9ae7c5e1796b4f90b2a31a6dea3c67f2c77d126b537a212c40711e4
```

查看容器中mysql的3306端口映射到的主机端口：

```
$ sudo docker port 643c9b48e9ae 3306
0.0.0.0:49171
```

可以看到，Docker将本地的49171端口映射到了test_mysql容器的3306端口。

启动apache-php容器：

```
$ sudo docker run --name test_apache-php -d -P -e MYSQL_ADDR=192.168.1.2:49171
lqkaixin/centos-apache-php:v1
45813120e2e77cc0a5d683f380788377e9aaaeeea3a4072d1573e4b119c6c13b
```

这里我们用-e重新指定了MySQL的主机地址192.168.1.2:49171，其中192.168.1.2是mysql容器所在的主机，端口为mysql容器的3306映射到主机上的端口，我们刚查询到为49171。

Apache服务器对外暴露了其80端口，我们需要查看它映射到了主机上的哪个端口：

```
$ sudo docker port 45813120e2e77cc 80
0.0.0.0:49172
```

这里是49172。稍后通过浏览器访问该端口，即可访问容器提供的Web服务。

查看我们的容器：

```
$ sudo docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED
STATUS              PORTS                                  NAMES
45813120e2e7        lqkaixin/centos-apache-php:v1"/bin/sh -c  /root/ru              4 minutes ago
Up 4 minutes        0.0.0.0:49172->80/tcp                  test_apache-php
643c9b48e9ae        lqkaixin/centos-mysql:v1"/bin/sh -c    /root/ru                 9 minutes ago
Up 9 minutes        0.0.0.0:49171->3306/tcp                test_mysql
```

可以看到，test_mysql和test_apache-php这两个容器已经正常运行了。打开浏览器，在地址栏中输入192.168.1.2:49172/test.php（此处IP和端口需要根据你的实际情况而定），如果显示如图6-2所示的界面，即证明容器正常工作了。

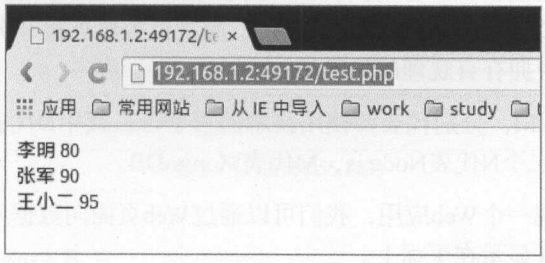


图6-2 Apache+PHP+MySQL展示页面

第7章 综合案例2：DLNNM

本章也是一个综合案例，旨在将前面所学的内容应用到实际的开发当中。通过实例的操作，我们将综合学习和巩固Dockerfile构建、容器连接、跨主机容器连接和分发镜像等知识，以此达到对Docker在实际应用中拥有直观理解的目的。

本案例取名为DLNNM，分别代表要使用技术的首字母，其中D代表Docker，L代表Linux，第一个N代表Nginx，第二个N代表Node.js，M代表MongoDB。

本案例的目标是实现一个Web应用，我们可以通过Web页面向数据库中插入一条记录，同时也能查询所有记录并将其显示在页面上。

案例的部署结构如图7-1所示，其中包含3台主机和5个容器。主机1运行着MongoDB服务和MongoDB代理容器，主机2运行着Node.js服务器和MongoDB访问代理容器，主机3运行着前端Web服务程序Nginx容器。其中代理容器并不是必需的，但是拥有它，架构会变得更灵活。当然，我们也可以完全将这3个Docker主容器（MongoDB、Node.js和Nginx）部署在同一台主机上，这由实际的业务需求所决定。根据实际的业务需求，我们还可以在这套结构里添加多个MongoDB容器服务或者其他种类的容器服务（如Redis、PHP等）。

本章包含以下内容：

- 介绍MongoDB数据库系统，并将其制作成镜像；
- 介绍Node.js开发平台，并以此开发node-web-api镜像；
- 使用代理容器连接MongoDB容器和Node.js容器；
- 基于Nginx服务器开发前端Web页面，并将其制作成镜像。

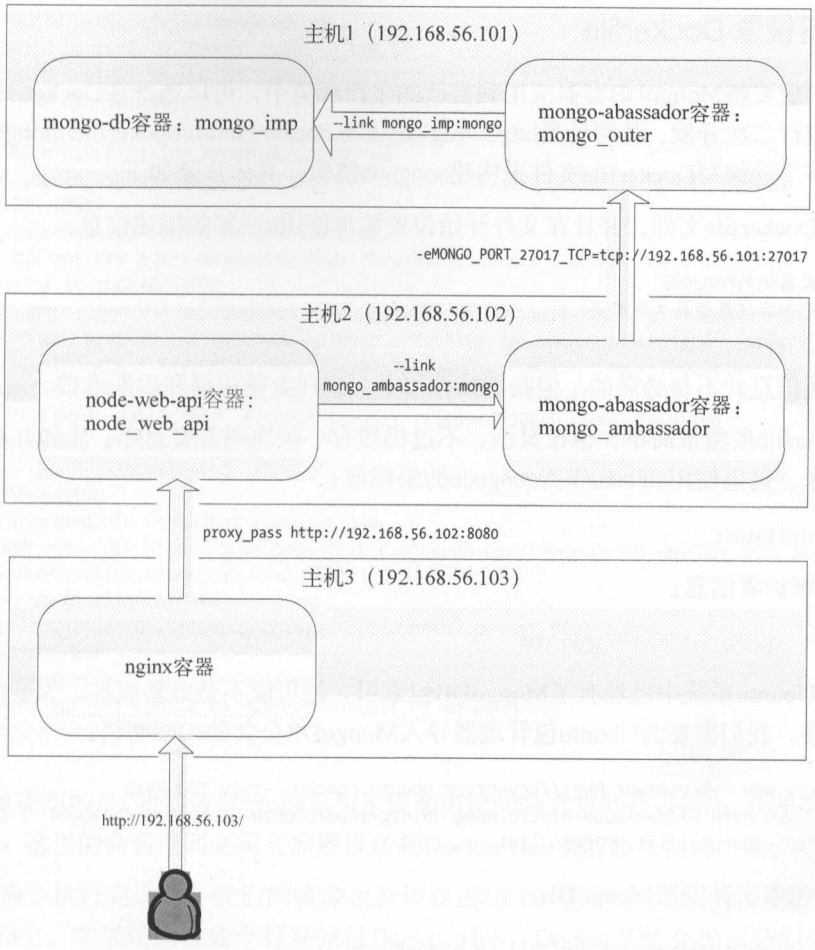


图7-1 Nginx+Node.js+MongoDB的部署架构

7.1 构建 mongodb 镜像

MongoDB是一款流行的开放源码的非关系型数据库系统（NoSQL），常用于大数据量、高并发、弱事务的互联网应用。MongoDB的官网地址是http://www.mongodb.com/。将MongoDB数据库系统容器化可以带来以下几个好处。

- 更容易维护。
- 启动速度快。
- 方便与他人进行分享。

本节将讲解如何使用Dockerfile来构建mongodb镜像。

7.1.1　编写镜像 Dockerfile

如果用户想要将MongoDB容器应用到自己的生产环境中，可以选择在Docker Hub的mongodb官方镜像上进行二次开发，其网址是https://registry.hub.docker.com/u/dockerfile/mongodb/。但在本节中，我们将手动编写Dockerfile文件来构建mongodb镜像，具体步骤如下。

(1) 创建Dockerfile文件，并且在文件开始位置添加使用#注释的描述信息：

```
# 名称：容器化的MongoDB
# 用途：用作后端数据持久化服务
# 创建时间：2015.01.15
```

这些描述信息并不是必需的，但我们推荐写上，这样方便传播和后期维护。

(2) Dockerfile虽然很简单，也很灵活，不过仍然有一些规则需要遵守，比如开头一定是定义根镜像的命令，这里使用ubuntu作为mongodb的根镜像：

```
FROM ubuntu:latest
```

(3) 声明维护者信息：

```
MAINTAINER xixihe xxh281weeks@qq.com
```

(4) 即使Ubuntu系统中已经有了MongoDB包索引，但可能不是最新版本。为了使用官方最新的包进行安装，我们需要为Ubuntu包管理器导入MongoDB公共的GPG密钥：

```
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee
    /etc/apt/sources.list.d/10gen.list
```

(5) 更新包索引并安装MongoDB：

```
RUN apt-get update && apt-get install -y mongodb-org
```

如果有需要，我们还可以分别为MongoDB的各个组件指定特定的版本，如下：

```
RUN apt-get update && apt-get install -y mongodb-org=2.6.1 mongodb-org-server=2.6.1
mongodb-org-shell=2.6.1 mongodb-org-mongos=2.6.1 mongodb-org-tools=2.6.1
```

(6) MongoDB默认的服务端口是27017，所以还要使用EXPOSE命令将这个端口映射到主机：

```
EXPOSE 27017
```

(7) 使用ENTRYPOINT命令告诉Docker在MongoDB容器启动时运行mongod服务：

```
ENTRYPOINT ["/usr/bin/mongod"]
```

7.1.2　构建和上传镜像

有了Dockerfile之后，进入Dockerfile文件所在的目录，然后使用build命令来构建镜像：

```
# docker build --tag xixihe/mongo-db:v1 .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
 ---> 8eaa4ff06b53
Step 1 : MAINTAINER xixihe xxh281weeks@qq.com
 ---> Running in 3f2b9923ab0b
 ---> c696d91f83af
Removing intermediate container 3f2b9923ab0b
Step 2 : RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
 ---> Running in 2bc22605302a
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir /tmp/tmp.oJW1Is5C5Q
--no-auto-check-trustdb --trust-model always --keyring /etc/apt/trusted.gpg --primary-keyring
/etc/apt/trusted.gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
gpg: requesting key 7F0CEB10 from hkp server keyserver.ubuntu.com
gpg: key 7F0CEB10: public key "Richard Kreuter <richard@10gen.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
 ---> da4588da3f6c
Removing intermediate container 2bc22605302a
Step 3 : RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee
/etc/apt/sources.list.d/mongodb.list
 ---> Running in c31b6e026afd
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen
 ---> 2291fe04794f
Removing intermediate container c31b6e026afd
Step 4 : RUN apt-get update && apt-get install -y mongodb-org
 ---> Running in 57c161eaf46a
...(略)
```

这里需要说明的是，我们通过tag标志为本镜像指定的命名空间是xixihe，仓库名为mongo-db，TAG标记为v1。这里的命名空间必须与需要推送到Docker Hub上的账号保持一致。

当build命令执行完毕时，我们的镜像也就构建成功了。接下来，通过push命令将镜像推送到Docker Hub上。如果还未在命令行登录过Docker Hub，Docker系统会提示我们先登录Docker Hub，此时直接输入账号和密码即可：

```
# docker push xixihe/mongo-db:v1
he push refers to a repository [xixihe/mongo-db:v1] (len: 1)
Sending image list
Please login prior to push:
Username: xixihe
Password:
Email: xxh281weeks@qq.com
Login Succeeded
The push refers to a repository [xixihe/mongo-db:v1] (len: 1)
Sending image list
Pushing repository xixihe/node-mongo (1 tags)
511136ea3c5a: Image already pushed, skipping
3b363fd9d7da: Image already pushed, skipping
...(略)
```

至此，本案例的数据持久mongodb镜像就已经创建好了，并且可以从Docker Hub中自由拉取。

7.2　构建Node.js镜像

Node.js是一个基于Google V8引擎建立的使用JavaScript语言编写的服务器开发平台，可用来快速构建易于扩展的网络应用。Node.js的非阻塞I/O的事件驱动模型机制使得它更轻量、更高效，特别适用于处理分布式设备密集实时数据。

本节中，我们使用Node.js开发一套简单的web-api服务，这套API主要用于对MongoDB进行增查操作，最后将其打包至安装了Node.js运行时的镜像中。

在开始之前，我们先创建工作目录node，这个目录用来存放Dockfile文件及其所依赖的项目文件及文件夹：

```
# mkdir node
# cd node
# touch Dockerfile
# mkdir src
```

这个目录包含Dockerfile文件和src目录，其中src目录用于存放项目的Node.js项目代码。

7.2.1　项目源文件

我们将Node.js Web应用的全部源码放入到src文件夹。一个Node.js应用一般由源文件和其所依赖的第三方模块组成。首先，在src文件夹里面建立项目的包文件package.json，这个包文件用来添加本Web应用的描述信息，包括名称、作者、版本以及项目所依赖的第三方库等信息。下面是包文件package.json的内容：

```
{
    "name": "Docker Node.js Web应用",
    "private": true,
    "version": "0.0.1",
    "description": "Docker Node.js Web应用",
    "author": "xixihe <xxh281weeks@qq.com>",
    "dependencies": {
        "express": "4.11.0",
        "mongodb": "1.4.28",
        "body-parser":"1.9.2"
    }
}
```

Node.js利用包管理器NPM来管理项目依赖关系。通过npm install命令就能导入在包文件package.json中字段dependencies声明的，第三方库到项目中。

我们的node-web应用共依赖3个包，下面简要介绍一下这3个包的作用。

- Express：Express是Node.js平台下的一个快速、灵活、极简的Web应用开发框架，它提供一系列强大的特性，扩展了Node.js原生的HTTP接口，帮助我们创建各种 Web 和移动设备应用。其官方网址是http://expressjs.com/。

- MongoDB：因为我们的node-web应用需要用到MongoDB数据库，所以还需要访问数据库的驱动程序，也就是mongodb，具体可参考文档：http://mongodb.github.io/ node-mongodb-native/contents.html。
- body-parser：express框架下解析HTTP请求数据的一个中间件，本案例使用此模块来解析通过POST上传的数据内容。关于该包的更多内容，可参考它的文档和源码：https://github.com/expressjs/body-parser。

接下来，在src文件夹里创建应用的入口文件index.js，其源代码如下：

```
var express = require('express');
var bodyParser = require('body-parser');
var mongo = require('mongodb');
var PORT = 8080;
//var db = new mongo.Db('nginx-node-mongo', new mongo.Server('localhost', 27017));
//var db = new mongo.Db('nginx-node-mongo', new mongo.Server('192.168.56.101', 27017));
var db = new mongo.Db('nginx-node-mongo', new mongo.Server('mongo', 27017));
var app = express() ;
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(bodyParser.raw());
app.get('/node-web-api/msg/find', function(req, res){
    console.log('get: ' + req.originalUrl);
    db.collection('msg_list').find().toArray(function(err, docs){
        if (err){
            res.send('-1');
        } else {
            var str = docs.length + '<br/>';
            for (var i = 0; i < docs.length; ++i){
                str += docs[i].time + '.' + docs[i].data + '<br/>';
            }
            res.send(str);
        }
    })
});

app.post('/node-web-api/msg/add', function(req, res){
    console.log('post: '+ req.originalUrl );
    if (!req.body || !req.body.item) {
        res.send('2');
        return ;
    }
    db.collection('msg_list').save({time: new Date().getTime(), data: req.body.item}, function(err,
        result){
        if (err) {
            res.send('1');
        } else {
            res.send('0');
        }
    });
});
db.open(function(err, db) {
```

```
    if (!err) {
        console.log('mongodb-server connected!');
        app.listen(PORT);
        console.log('node-web-server started! port: ' + PORT);
    }
});
```

下面简要说明该源文件。应用首先连接到mongo主机在端口27017开启的MongoDB服务，连接成功后将在8080端口监听web-api的调用。27017是MongoDB默认的监听端口，我们使用常量字符串mongo作为MongoDB服务的主机名，这依赖于7.3.3节中讲到的--link参数设置。实际上，mongo是指我们的代理容器。如果不使用--link参数，则可以直接使用域名或者IP，即源代码文件中通过#号注释掉的两行。

本案例只实现了两个web-api接口，一个是向数据库中添加一条用户上传的内容：

```
POST http://host/node-web-api/msg/add
```

另一个是从MongoDB数据库中查询所有用户上传的内容：

```
GET http://host/node-web-api/msg/find
```

7.2.2　编写镜像 Dockerfile

Dockerfile所依赖的项目文件准备就绪之后，接下来开始编写Dockerfile文件。以下是Dockerfile文件的全部内容：

```
# 名称：容器化的node-web-api应用
# 用途：实现CRUD MongoDB的Web API
# 创建时间：2015.01.19
#指定基础镜像
FROM ubuntu:latest
#安装curl
RUN apt-get update
RUN apt-get install -y curl
#更新Node安装源
RUN curl -sL https://deb.nodesource.com/setup | sudo bash -
RUN apt-get update
#安装Node.js和NPM
RUN apt-get install -y nodejs
RUN apt-get install -y npm
#复制项目源文件到镜像
COPY ./src /src
#进入项目源文件目录，使用NPM安装项目依赖库
RUN cd /src; npm install;
#暴露项目所监听的端口
EXPOSE 8080
#定义项目执行入口
CMD ["nodejs", "/src/index.js"]
```

7.2.3 构建和上传镜像

接下来，使用build命令设置镜像，并将其命名为xixihe/node-web-api:v1：

```
# docker build --tag xixihe/node-web-api:v1 .
...(略)
```

因为Dockerfile文件里牵涉到几个比较耗时的网络操作，读者需要耐心等待构建过程。

创建完毕后，通过docker images命令验证镜像：

```
# docker images
REPOSITORY              TAG    IMAGE ID       CREATED              VIRTUAL SIZE
xixihe/node-web-api     v1     17beb304695d   25 hours ago         379.5 MB
...(略)
```

和前面一样，我们将镜像提交到Docker Hub上：

```
# docker push xixihe/node-web-api:v1
...(略)
```


7.3 连接 Node.js 服务和 MongoDB 服务

我们已经在4.3.3节中介绍过跨主机使用代理容器来连接容器。本节中，我们将这种模式应用到Node.js与MongoDB连接的案例中。

7.3.1 制作代理镜像 mongo-abassador

我们的代理镜像基于busybox，不过不能直接使用busybox做基础镜像。因为还需要用到socat等busybox没有的工具，所以首先要使用制作代理镜像的基础镜像docker-ut。制作docker-ut的脚本文件mkimage-unittest.sh改自https://github.com/docker/docker/bl ob/master/contrib/mkimage-unittest.sh。修改后的源码如下：

```
#!/usr/bin/env bash
# Generate a very minimal filesystem based on busybox-static,
# and load it into the local docker under the name "docker-ut".
missing_pkg() {
    echo "Sorry, I could not locate $1"
    echo "Try 'apt-get install ${2:-$1}'?"
    exit 1
}
BUSYBOX=$(which busybox)
[ "$BUSYBOX" ] || missing_pkg busybox busybox-static
SOCAT=$(which socat)
[ "$SOCAT" ] || missing_pkg socat
shopt -s extglob
set -ex
ROOTFS=`mktemp -d ${TMPDIR:-/var/tmp}/rootfs-busybox.XXXXXXXXXX`
```

```
trap "rm -rf $ROOTFS" INT QUIT TERM
cd $ROOTFS
mkdir bin etc dev dev/pts lib proc sys tmp
touch etc/resolv.conf
cp /etc/nsswitch.conf etc/nsswitch.conf
echo root:x:0:0:root:/:/bin/sh > etc/passwd
echo daemon:x:1:1:daemon:/usr/sbin:/bin/sh >> etc/passwd
echo root:x:0: > etc/group
echo daemon:x:1: >> etc/group
ln -s lib lib64
ln -s bin sbin
cp $BUSYBOX $SOCAT bin
for X in $(busybox --list)
do
    ln -s busybox bin/$X
done
rm bin/init
ln bin/busybox bin/init
cp -P /lib/x86_64-linux-gnu/lib{pthread*,c*(-*),dl*(-*),nsl*(-*),nss_*,util*(-*),wrap,z}.so* lib
cp /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 lib
cp -P /lib/x86_64-linux-gnu/lib{crypto,ssl}.so* lib
cp -P /usr/lib/x86_64-linux-gnu/lib{crypto,ssl}.so* lib
for X in console null ptmx random stdin stdout stderr tty urandom zero
do
    cp -a /dev/$X dev
done
chmod 0755 $ROOTFS # See #486
tar --numeric-owner -cf- . | docker import - docker-ut
docker run -i -u root docker-ut /bin/echo Success.
rm -rf $ROOTFS
```

接下来，使用chmod赋予mkimage-unittest.sh文件可执行权限，然后以root权限运行该脚本，此时docker-ut就生成好了。操作如下：

```
$ chmod +x mkimage-unittest.sh
$ sudo ./mkimage-unittest.sh
...(略)
$ sudo docker images
REPOSITORY            TAG     IMAGE ID       CREATED             VIRTUAL SIZE
xixihe/ docker-ut     v1      c3018ec164d8   34 minutes ago      7.23 MB
...(略)
```

如果运行mkimage-unittest.sh的过程中，提示类似下面的信息：

```
“Sorry, I could not locate socat,
Try 'apt-get install socat'?”
```

我们就要按照提示在主机上安装socat或其他缺失的组件，然后再重新运行mkimage-unittest.sh。

此外，运行mkimage-unittest.sh时可能还会出现如下错误：

```
...(略)
/lib/x86_64-linux-gnu/libwrap.so.0.7.6 /lib/x86_64-linux-gnu/libz.so.1
/lib/x86_64-linux-gnu/libz.so.1.2.8 lib
```

```
+ cp /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 lib
+ cp -P '/usr/lib/x86_64-linux-gnu/libcrypto.so*' '/usr/lib/x86_64-linux-gnu/libssl.so*' lib
cp: cannot stat '/usr/lib/x86_64-linux-gnu/libcrypto.so*': No such file or directory
cp: cannot stat '/usr/lib/x86_64-linux-gnu/libssl.so*': No such file or directory
...(略)
```

对于上面的错误，我们需要安装libssl1.0.0和libssl-dev来解决：

```
$ sudo apt-get install libssl1.0.0
$ sudo apt-get install libssl-dev
```

通过docker images命令确保docker-ut创建成功后，接下来编写构建代理镜像的Dockerfile，全部内容如下：

```
FROM    docker-ut
MAINTAINER  xxh281weeks@qq.com
CMD     env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat
TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/'  | sh && top
```

如同前两节，我们使用build命令构建镜像，构建成功后使用push命令将其上传至Docker Hub：

```
# docker build --tag xixihe/mongo-abassador:v1 .
...(略)
# docker push xixihe/mongo-abassador:v1
...(略)
```

7.3.2 启动 MongoDB 服务

本案例的MongoDB数据库服务由两个容器组成，一个是主容器MongoDB，另一个是对外提供服务的代理容器。MongoDB服务将部署在主机1（192.168.56.101）上。接下来，我们按步骤启动Mongo服务容器及对外服务的代理容器。

(1) 参照第1章，在主机1上安装Docker服务。如果已经安装，则略过此步骤。

(2) 从Docker Hub拉取7.1节中所创建的mongo-db:v1镜像及代理镜像mongo-abassador:v1。如果镜像是在本机创建的，则可以略过这一步。

```
# docker pull xixihe/mongo-db:v1
...(略)
# docker pull xixihe/mongo-abassador:v1
...(略)
```

(3) 以守护进程模式启动Mongo服务，取名为mongo_imp：

```
# docker run -d --name mongo_imp xixihe/mongo-db:v1
```

(4) 运行代理容器，使用--link参数连接mongo，并指定对外服务的映射端口27017：

```
# docker run -d -p 27017:27017 --name mongo_outer --link mongo_imp:mongo
xixihe/mongo-abassador:v1
```

我们选择一台安装了mongo-client且能连接上主机1的机器来测试MongoDB服务是否工作正常，如果显示如下，就表明工作正常：

```
# mongo 192.168.56.101
MongoDB shell version: 2.6.5
connecting to: 192.168.56.101/test
> show dbs
admin              (empty)
local              0.078GB
>
```

现在，我们的主机1就已经具备了对外提供MongoDB数据库服务的功能了。需要注意的是，本案例的MongoDB并没有加入账号管理功能。为了保证数据库的安全性，读者可以参照MongoDB文档加入账号访问控制功能。

7.3.3　启动 Node-Web-API 服务

Node-Web-API服务将部署在主机2（192.168.56.102）上面，该服务也由两个容器组成，其中一个是访问MongoDB服务的代理容器，另外一个是连接代理容器且提供web-api调用的Node.js容器。接下来，逐步演示启动服务的过程。

(1) 参照第1章，在主机2上安装Docker服务。如果已经安装，则略过此步骤。

(2) 从Docker Hub上拉取7.2节创建的node-web-api:v1镜像和7.3.1节创建的镜像，如果镜像已经存在本机，则略过此步骤：

```
# docker pull xixihe/node-mongo:v1
# docker pull xixihe/mongo-abassador:v1
```

(3) 启动连接数据库服务的代理容器，命名为mongo_ambassador：

```
#docker run -d --name mongo_ambassador --expose 27017 -e
MONGO_PORT_27017_TCP=tcp://192.168.56.101:27017 xixihe/mongo-abassador:v1
#
```

-e参数指定的环境变量MONGO_PORT_27017_TCP=tcp://192.168.56.101:27017中的IP地址是在主机1的IP地址，端口是主机1上mongo_outer容器对外监听的端口。按照上面的方法创建的代理容器会在内部根据这个环境变量建立与主机1上的mongo_outer代理关系，本主机的其他应用通过mongo_ambassador容器就可以访问到主机1上mongo_imp容器所提供的数据服务。

(4) 启动Node-Web-API容器，命名为node_web_api：

```
# docker run -d --name node_web_api -p 8080:8080 --link mongo_ambassador:mongo
xixihe/node-web-api:v1
```

node-web-api应用会在8080端口监听Web API调用，我们使用-p 8080:8080直接将这个端口映射到主机上。同时，我们使用--link参数将mongo_ambassador以别名mongo映射到容器内，这意

味着在node_web_api容器内能直接以主机名mongo来访问mongo_amb assador。

我们的node-web-api服务已经准备就绪了。现在使用curl来测试一下web-api命令：

```
# curl http://192.168.56.102:8080/node-web-api/msg/find
5<br/>1421755056144.abc<br/>1421755061730.afdgafasfasd<br/>1421755075377.这是测试
1<br/>1421755079585.这是测试2<br/>1421755084285.测试完毕<br/>...(略)
```

返回的内容是我预先加入数据库的数据。

7.4 搭建前端 Nginx

Nginx是一款支持HTTP、HTTPS、SMTP、POP3、IMAP等协议的反向代理服务器，也常用作负载均衡、HTTP缓存和Web服务器。由于它开源、配置简单以及拥有高并发、高性能、低内存、稳定性高等诸多优良特性，目前已经成为互联网上应用最广泛的Web服务器之一。

本案例以Nginx作为前端服务器，主要提供两方面的功能：

- 提供静态页面的访问服务；
- node-web-api反向代理服务。

7.4.1 构建镜像并运行

接下来，我们将逐步讲解如何开发和部署这样的Nginx容器服务。在Docker Hub上已经托管了Nginx的官方镜像，其地址是https://registry.hub.docker.com/_/nginx/，我们直接使用这个镜像来部署应用。

(1) 如果主机3还没有安装Docker服务，先参照第1章的方法在主机3上安装Docker。

(2) 搜索并拉取官方的Nginx镜像：

```
# docker search nginx
NAME                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
nginx               Official build of Nginx.                        507                  [OK]
jwilder/nginx-proxy Automated Nginx reverse proxy for docker c...   112                  [OK]
dockerfile/nginx    Trusted automated Nginx (http://nginx.org/...   92
...(略)
# docker pull nginx
...(略)
# docker images
REPOSITORY   TAG       IMAGE ID        CREATED       VIRTUAL SIZE
nginx        latest    1822529acbbf    2 weeks ago   91.64 MB
...(略)
```

(3) 在主机3上的home目录下创建一个名为nginx-node的文件夹，该文件夹用于存放Web静态文件和自定义的Nginx配置文件server.conf。

(4) 自定义nginx配置。server.conf自定义的内容如下：

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;
    root /home/node/www;
    #root /usr/share/nginx/html;
    index index.html index.htm;

    charset utf-8;
    server_name localhost;

    location /node-web-api/ {
        proxy_pass http://192.168.56.102:8080;
    }

    location / {
        try_files $uri $uri/ =404;
    }
}
```

这个自定义配置文件用来替换原生nginx容器里的默认配置，我们会在启动容器时通过-v参数来设置这种替换。

另外，我们还需要关注两个点。一是我们配置了nginx的root目录为/home/node/www，该目录是容器里web服务的根目录，在启动容器时需要通过-v参数指定主机目录到该目录的映射。另一个是我们通过proxy_pass将node-web-api的请求全部反向代理到http://192.168.56.102:8080地址。不难发现，这个http地址就是7.2节在主机2上面构建Node.js Web服务的地址。

(5) 在nginx-node文件夹下创建www文件夹，这个文件夹用于存放Nginx对外服务的静态文件：

```
~/nginx-node/www # ls -l
total 92
-rwxrwxrwx 1 xxh xxh    29  1月 15 16:12 index.html
-rwxrwxrwx 1 xxh xxh 84320  12月 18 23:17 jquery-2.1.3.min.js
```

其中有两个文件，具体如下所示。

- jquery-2.1.3.min.js：jQuery 是一个兼容多浏览器的JavaScript前端框架，其官方网址是http://jquery.com/。本例使用它来简化对node-web-api的异步调用。
- index.html：本案例应用的主页面，它实现了对node-web-api添加和查询API的调用。源码如下：

```
<!DOCTYPE>
<html>
  <meta charset='utf-8'/>
  <head>
      <script type="text/javascript" src="./jquery-2.1.3.min.js"></script>
      <script type="text/javascript">
          $(document).ready(function() {
```

```
            $('#ADD').click(function(){
                var obj = {item: $('#item').val()};
                $.post("/node-web-api/msg/add", obj, function(data, status){
                    alert("add result: " + status);
                });
            });
            $('#FIND').click(function(){
                $.get("/node-web-api/msg/find", function(data, status){
                    $('#content').html(data);
                });
            });
        });
    </script>
</head>
<body>
    <input id='item' type='text' ><input  id='ADD' type='button' value='添加'><br/>
    <input id='FIND' type='button' value='查询'>
    <p id="content"></p>
</body>
</html>
```

界面预览效果如图7-2所示。

图7-2 index.html界面

(6) 在主机3以守护模式启动Nginx容器服务，具体为：

```
# docker run -d -v /home/xinhua/nginx/server.conf:/etc/nginx/conf.d/default.conf:ro -v
/home/xinhua/nginx/www:/home/node/www:ro -p 80:80 nginx
```

首先，我们用-v参数将自定义的server.conf替换容器的默认配置default.conf，然后将主机的页面文件夹映射到server.conf所配置的容器文件夹/home/node/www。这台主机最终是直接提供给外网服务的，所以我们将容器内的80端口直接映射到外网。

7.4.2 验证 Web 应用

现在，Web应用已经完整部署好了，我们可以选择在任何一台能连接主机3的机器上通过http链接http://192.168.56.103/来访问该Web应用了，如图7-3所示。

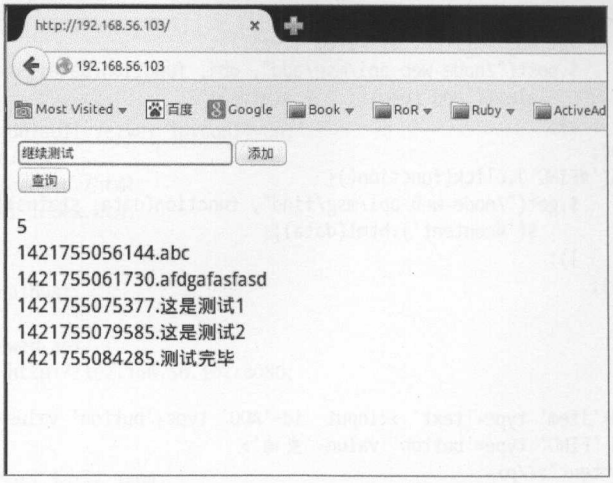


图7-3　Web应用展示

我们可以持续不断地添加内容到数据库中，并且通过“查询”按钮查询所有数据并将其显示出来。而且下次再访问这个页面时，还仍然能查询到以前添加的数据。

让我们回到主机3，查看Nginx运行日志：

```
# docker logs 0456798a89c9
192.168.56.102 - - [20/Jan/2015:11:51:25 +0000] "GET / HTTP/1.1" 200 736 "-" "Mozilla/5.0 (X11; Ubuntu;
    Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0" "-"
192.168.56.102 - - [20/Jan/2015:11:51:25 +0000] "GET /jquery-2.1.3.min.js HTTP/1.1" 200 84320
    "http://192.168.56.103/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101
    Firefox/31.0" "-"
192.168.56.102 - - [20/Jan/2015:11:51:25 +0000] "GET /favicon.ico HTTP/1.1" 404 168 "-" "Mozilla/5.0
     (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0" "-"
192.168.56.102 - - [20/Jan/2015:11:51:25 +0000] "GET /favicon.ico HTTP/1.1" 404 168 "-" "Mozilla/5.0
     (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0" "-"
...(略)
192.168.56.102 - - [20/Jan/2015:11:57:58 +0000] "POST /node-web-api/msg/add HTTP/1.1" 200 1
    "http://192.168.56.103/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101
    Firefox/31.0" "-"
192.168.56.102 - - [20/Jan/2015:11:58:03 +0000] "POST /node-web-api/msg/add HTTP/1.1" 200 1
    "http://192.168.56.103/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101
    Firefox/31.0" "-"
192.168.56.102 - - [20/Jan/2015:11:58:05 +0000] "GET /node-web-api/msg/find HTTP/1.1" 200 154
    "http://192.168.56.103/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101
    Firefox/31.0" "-"
```

该日志显示了Nginx所处理的静态资源请求和node-web-api的调用过程。

使用同样的方法，我们可以在主机1和主机2上分别查看MongoDB与Node.js的运行日志。这里就不一一演示了，读者可以自行查看。

第8章 阿里云Docker开发实践

作为2014年兴起的技术，Docker技术发展迅猛，国外各大云计算公司及软件公司（如Google、Amazon、微软等巨头）纷纷加入Docker阵营，而国内的BAT三大云平台也不例外，积极作出响应，开始支持Docker。

阿里云作为国内最著名的云平台之一，越来越多的中小企业及创业公司使用它来部署自己的应用。本章将以阿里云为平台，并结合Docker技术本身的特性，介绍在阿里云上进行Docker开发的一般流程。

图8-1对比了传统的软件开发、测试、部署与使用Docker进行开发的不同。流程大体一致，但项目相关者的关注点从底层项目工程转移到更单一的镜像上来。使用镜像来打包应用，可以减少运维的工作，并且测试人员和运维人员都不需要关注更多项目的底层细节，只需要将精力集中在镜像本身上即可。当然，前提是开发人员、测试人员和运维人员都需要对Docker原理及操作有一定的了解。显然，这种付出是值得的。

本章所讲的知识不仅仅只是应用在阿里云上，它还能作为一般开发流程的指导。本章主要包含以下内容：

- 在ECS上部署Docker服务；
- 在ECS上部署存储镜像在OSS上的私有注册服务器；
- 介绍开发人员、测试人员和运维人员在使用Docker协同工作时需要注意的地方。

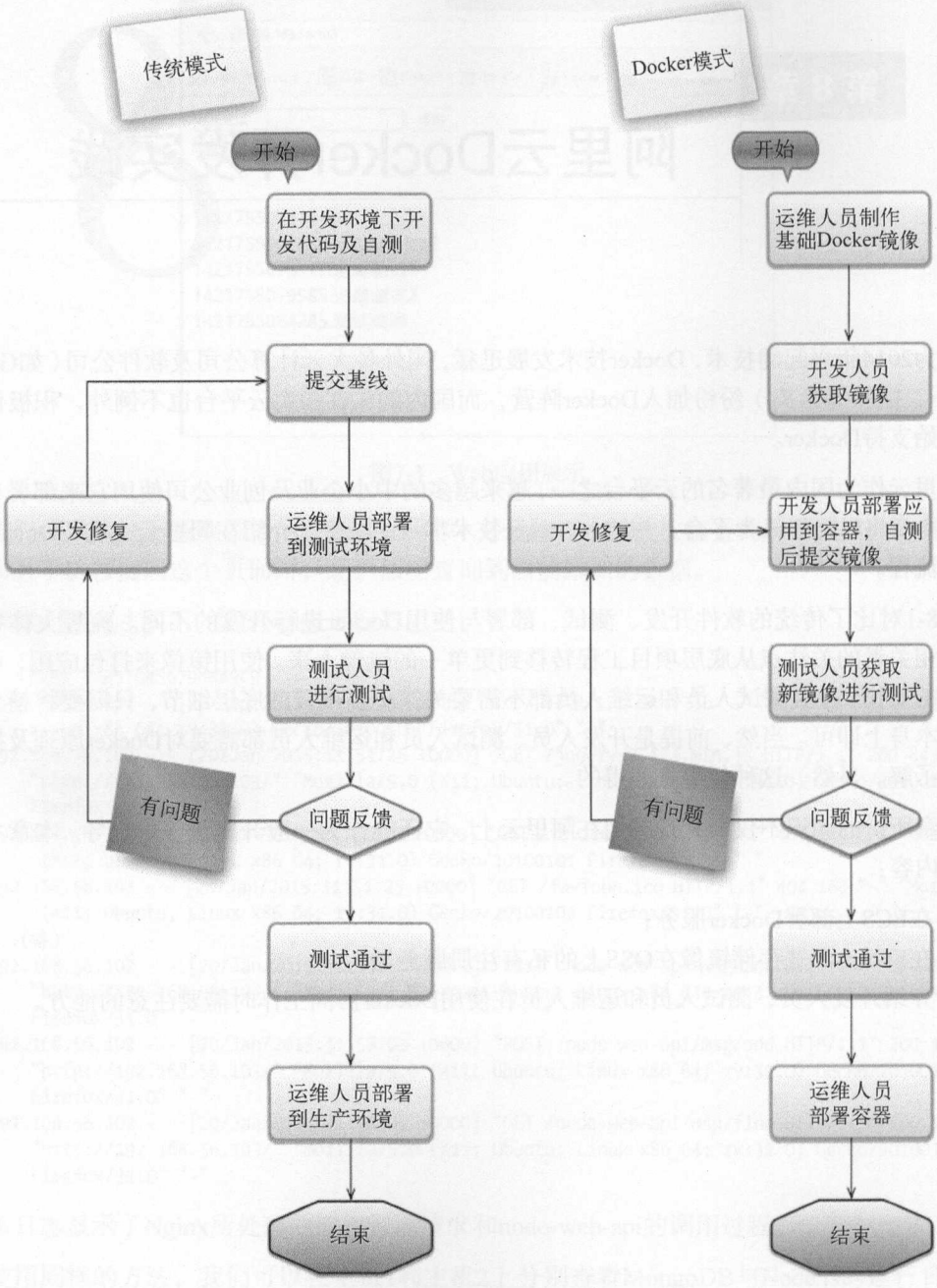


图8-1　Docker流程与传统流程对比（此图摘自阿里云官网）

8.1 阿里云 Docker 介绍

作为当前最炙手可热的容器技术，阿里云已经支持在ECS上部署Docker容器应用。我们可以在ECS上把应用打包成Docker镜像，运行Docker容器，也可以从阿里云提供的镜像库中快速下载Docker Hub官方镜像，还可以在上面部署自己的私有镜像库，并和团队成员分享和协作。图8-2展示了阿里云ECS的Docker生态系统。

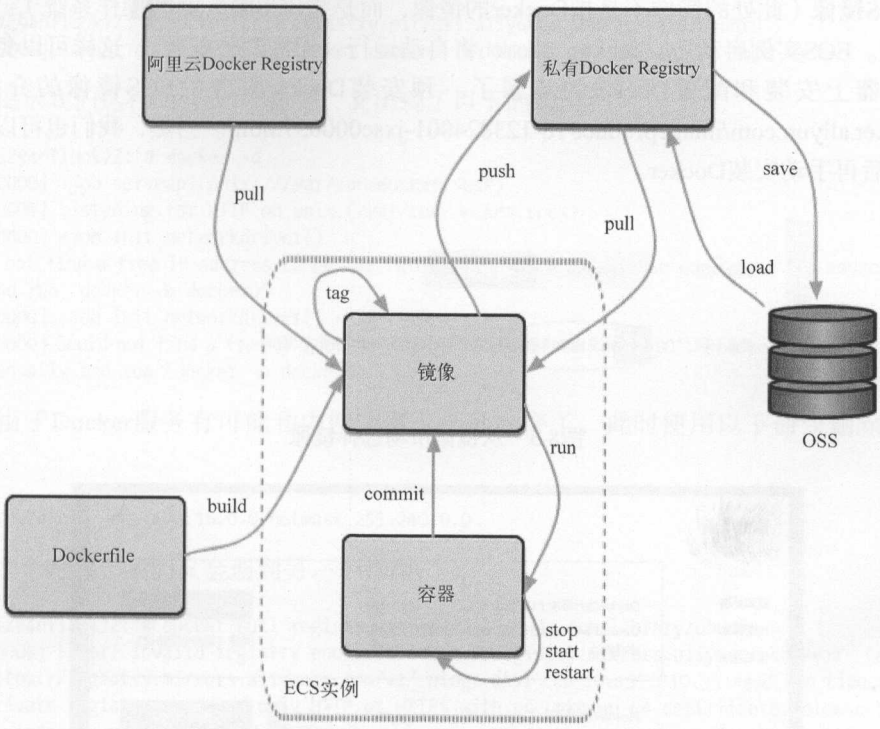


图8-2 阿里云ECS的Docker生态系统（此图摘自阿里云官网）

为了方便阿里云的Docker用户使用Docker的官方镜像，阿里云已经将Docker官方镜像库同步到国内服务器，这些镜像全部来自Docker Hub的用户stackbrew的镜像库，该用户的Docker Hub地址是https://hub.docker.com/u/stackbrew/。这里的镜像一部分由官方维护，一部分由软件官方社区维护。在ECS机器上使用pull命令可以快速拉取到镜像，从而避免直接去Docker Hub下载消耗的时间。现在支持的镜像的有debian、hello-world、zend-php、wordpress、ubuntu-upstart、ubuntu-debootstrap、ubuntu、ruby、registry、redis、rails、python、postgres、php、perl、opensuse、node、mageia、jruby、jenkins、java、hylang、hipache、golang、gcc、fedora、docker-dev、crux、crate、clojure、cirros、centos、busybox、buildpack-deps、nginx、mongo、neurodebian和mysql。使用这些镜像时，需要在镜像名前添加域registry.mirrors.aliyuncs.com。

通过pull命令下载镜像：

```
# docker pull registry.mirrors.aliyuncs.com/library/ubuntu
```

在Dockerfile中使用aliyun镜像库中的镜像作为根镜像：

```
FROM registry.mirrors.aliyuncs.com/library/ubuntu
```

我们在阿里云官网购买云服务器ECS时，可以在镜像市场为ECS服务器选择原生支持Docker服务的ECS镜像（此处的镜像不是指Docker的镜像，而是指ECS服务器的操作系统），如图8-3和图8-4所示。ECS实例启动后，docker daemon将自动运行，无需额外配置，这样可以免去手动在ECS服务器上安装和配置Docker的步骤了。预安装Docker服务的ECS镜像的介绍页面在http://market.aliyun.com/imageproduct/16-123824001-jxsc000057.html。当然，我们也可以选择公共镜像，然后再手动安装Docker。

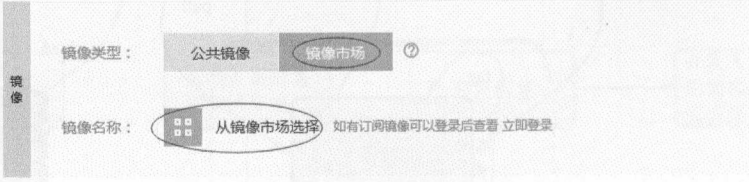


图8-3　从镜像市场选择镜像

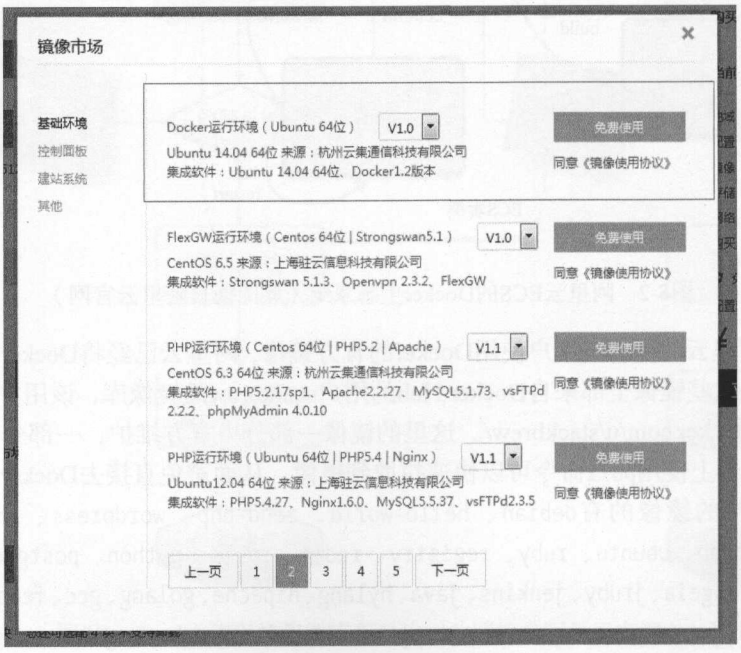


图8-4　选择预先安装配置好Docker的ECS实例镜像

如果没有使用预安装Docker的镜像，也可以手动安装。我们使用SSH登录到阿里云的ECS主机，然后按照下面的步骤进行安装：

```
root@iZ94nflqok7Z:~# apt-get install curl
root@iZ94nflqok7Z:~# curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

Docker安装成功后，接着尝试去从阿里云部署的镜像库拉取ubuntu:latest镜像，此时会出现下面的错误：

```
root@iZ94nflqok7Z:~# docker pull registry.mirrors.aliyuncs.com/library/ubuntu
FATA[0000] Cannot connect to the Docker daemon. Is 'docker -d' running on this host?
```

按照提示我们运行docker -d命令，又出现了以下的错误：

```
root@iZ94nflqok7Z:~# docker -d
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] +job init_networkdriver()
Could not find a free IP address range for interface 'docker0'. Please configure its address manually
    and run 'docker -b docker0'
INFO[0000] -job init_networkdriver() = ERR (1)
FATA[0000] Could not find a free IP address range for interface 'docker0'. Please configure its address
    manually and run 'docker -b docker0'
```

这是由于Docker服务有可能和内网的网卡地址冲突了，此时使用以下命令删除冲突地址即可：

```
sudo route del -net 172.16.0.0 netmask 255.240.0.0
```

可是这个时候，我们又会遇到另一个错误：

```
root@iZ94nflqok7Z:~# docker pull registry.mirrors.aliyuncs.com/library/ubuntu
FATA[0005] Error: Invalid registry endpoint https://registry.mirrors.aliyuncs.com/v1/: Get
    https://registry.mirrors.aliyuncs.com/v1/_ping: dial tcp 10.157.230.35:443: i/o timeout. If this
    private registry supports only HTTP or HTTPS with an unknown CA certificate, please add
    `--insecure-registry registry.mirrors.aliyuncs.com` to the daemon's arguments. In the case of HTTPS,
    if you have access to the registry's CA certificate, no need for the flag; simply place the CA
    certificate at /etc/docker/certs.d/registry.mirrors.aliyuncs.com/ca.crt
```

此时我们需要将registry.mirrors.aliyuncs.com这个域加入到非安全通信域。修改/etc/default/docker文件，添加--insecure-registry registry.mirrors.aliyuncs.com到DOCKER_OPTS，如下：

```
DOCKER_OPTS="--insecure-registry registry.mirrors.aliyuncs.com"
```

然后重启Docker服务即可：

```
root@iZ94nflqok7Z:~# service docker  restart
stop: Unknown instance:
docker start/running, process 8646
root@iZ94nflqok7Z:~# docker pull registry.mirrors.aliyuncs.com/library/ubuntu:latest
Pulling repository registry.mirrors.aliyuncs.com/library/ubuntu:latest
```

```
...(略)
root@iZ94nflqok7Z:~# docker images
REPOSITORY                                          TAG     IMAGE ID      CREATED       VIRTUAL SIZE
registry.mirrors.aliyuncs.com/library/ubuntu latest b39b81afc8ca  9 days ago     188.3 MB
```

我们发现使用registry.mirrors.aliyuncs.com注册服务器拉取镜像的速度要比到Docker Hub上拉取镜像的速度快很多。

8.2　部署镜像注册服务器

为了方便项目相关人员协作开发和共享镜像，企业或组织可以使用Docker Hub作为自己的私有镜像注册服务器，也可以在阿里云ECS云服务器上部署自己的私有镜像注册服务器。作为国内用户，后者是更值得推荐的一种方法。目前，官方的docker-registry镜像尚不支持阿里云的OSS存储。幸运的是，个人贡献者Chris给docker-registry专门开发了支持阿里云OSS的驱动，并将其制作成了Docker镜像。使用这个注册服务器镜像，我们已经能轻松地将镜像存储到阿里云OSS上了。本镜像的Docker Hub网址是https://registry.hub.docker.com/u/chrisjin/registry/。

OSS（Open Storage Service，开放存储服务）是阿里云对外提供的海量、安全和高可靠的云存储服务，如图8-5所示。鉴于OSS在存储方面的诸多优良特性，在搭建私有注册服务器时，我们也推荐使用OSS作为镜像的存储服务。阿里云OSS服务的开通网址为http://www.aliyun.com/product/oss/。

图8-5　阿里云OSS

开通OSS服务之后，我们会得到OSS_BUCKET、STORAGE_PATH和OSS_SECRET。使用支持OSS的docker-registry，我们就能轻松地在ECS上部署存储在OSS上的注册服务器：

```
docker run -e OSS_BUCKET=<your_ali_oss_bucket> -e STORAGE_PATH=/docker/ -e
OSS_KEY=<your_ali_oss_key> -e OSS_SECRET=<your_ali_oss_secret> -p 5000:5000 -d
chrisjin/registry:ali_oss
```

如果不使用chrisjin/registry:ali_oss，而是从https://github.com/docker/docker-registry下载官方的docker-registry，通过下面的步骤也能轻松支持OSS。

通过pip命令安装OSS驱动：

```
pip install docker-registry-driver-alioss
```

然后配置config.yml：

```
local: &local
    <<: *common
    storage: alioss
    storage_path: _env:STORAGE_PATH:/devregistry/
    oss_bucket: _env:OSS_BUCKET[:default_value]
    oss_accessid: _env:OSS_KEY[:your_access_id]
    oss_accesskey: _env:OSS_SECRET[:your_access_key]
```

最后，启动docker-registry：

```
DOCKER_REGISTRY_CONFIG= [your_config_path]  gunicorn -k gevent -b 0.0.0.0:5000 -w 1
docker_registry.wi:application
```

这样，我们的阿里云私有注册服务器就搭建好了，项目相关人员可以使用它分发和共享镜像了。

8.3 开发

注册服务器准备就绪之后，接下来开发人员就可以根据项目需求进入开发阶段了。在项目开发阶段，开发工程师可以使用ECS或者自己的机器作为开发机和测试，但是开发完之后要以镜像的方式交由测试人员去做测试。

8.3.1 项目开发

我们仍然以Node.js作为运行时、以Express作为Web框架，做一个简单的用作演示的静态HTML页面。关于Node.js和Express的介绍，读者可参考7.2节。

首先，创建项目文件夹AliNode，它共由两个文件组成。

- **服务器入口文件index.js。其源码如下：**

```
var express = require('express');
var app = express() ;
var PORT = 8080;
app.get('/node-web-api/show', function(req, res){
```

```
    var str = "<h1>你好，多克！</h1>"
    res.send(str);
});
app.listen(PORT);
```

上面的源码仅仅实现了一个/node-web-api/show接口。

- Node包文件package.json。其源码如下：

```
{
  "name": "Docker-Node-Web-APP",
  "private": true,
  "version": "0.0.1",
  "description": "Docker Node Web APP",
  "author": "xixihe <xxh281weeks@qq.com>",
  "dependencies": {
      "express": "4.11.0",
  }
}
```

8.3.2　制作和上传镜像

开发人员在自己的机器或者ECS服务器上完成开发之后，接下来应该将项目打包成镜像并自测它。一切就绪后，就可以将镜像推送至服务器上，并向测试工程师发出提测请求。提测请求中应该写明该镜像的修改点、启动方式、镜像存储在镜像服务器的位置及项目相关的信息。

进入AliNode文件夹，编写Dockerfile文件，全部内容如下：

```
# 定义基础镜像
FROM registry.mirrors.aliyuncs.com/library/ubuntu:latest
# 安装Node.js
RUN apt-get update
RUN apt-get install curl
RUN curl -sL https://deb.nodesource.com/setup | sudo bash -
RUN apt-get update
RUN apt-get install -y nodejs
RUN apt-get install -y npm
#复制项目源文件到镜像
COPY ./index.js /src
COPY ./package.json /src
#进入项目源文件目录，使用NPM安装项目依赖库
RUN cd /src; npm install;
#暴露项目所监听的端口
EXPOSE 8080
#定义项目执行入口
CMD ["nodejs", "/src/index.js"]
```

使用build命令构建镜像并上传至我们的私有注册服务器上：

```
# docker build -t 120.24.159.50:5000/node-web-api:v1 .
...(略)
```

```
# docker images
# docker build push 120.24.159.50:5000/node-web-api:v1
...(略)
```

8.4 测试

在开始测试之前，测试工程师首先需要确保自己的测试机上已经安装了Docker并且它正处于运行状态。在必要的情况下，需要保证Docker的版本与最终的生产环境一致。相关操作可以参考第1章。

由于Docker的跨平台特性，理论上测试工程师可以使用Windows、Linux（Ubuntu和CentOS等）、Mac OS中的任意一个作为测试平台，但为了保证与最终生产网络环境的一致性，最终仍有必要使用ECS机器作为测试机。

测试环境搭建好后，根据测试请求里说明的镜像地址拉取镜像，并按要求运行，根据镜像的目的测试所实现的业务。

如果在测试的过程中发现bug或不符合需求，应该尽快反馈给开发人员。开发人员修正后，重新将镜像推送到注册服务器，测试人员从镜像库拉取最新修改的镜像继续测试。反复几轮后，直到达到可发布的版本。最后，测试人员发布测试合格报告，并注明最终的镜像版本。

如果有多个测试工程师同时测试，各自使用自己的测试容器，还能保证测试之间不被干扰。

8.5 部署

运维工程师在收到测试工程师的测试报告及发布请求后，使用SSH登录到目标发布ECS主机。首先，运维工程师需要验证阿里云弹性机器是否安装了Docker服务、Docker服务版本是否符合需求、Docker服务是否已经启动，具体操作可参考第1章。接下来，从私有镜像注册服务器拉取测试报告里注明的最终测试通过的镜像，并运行它。此时，我们的node-web-api服务就已经成功发布到了外网。

由于某些原因，我们不再需要对外提供web-api服务了，运维工程师只要使用`docker stop`命令停止运行的容器就行了。同时，在有需要的情况下，运维人员也能使用`docker start`快速从容器中恢复服务。图8-6展示了本章实践的大致流程。

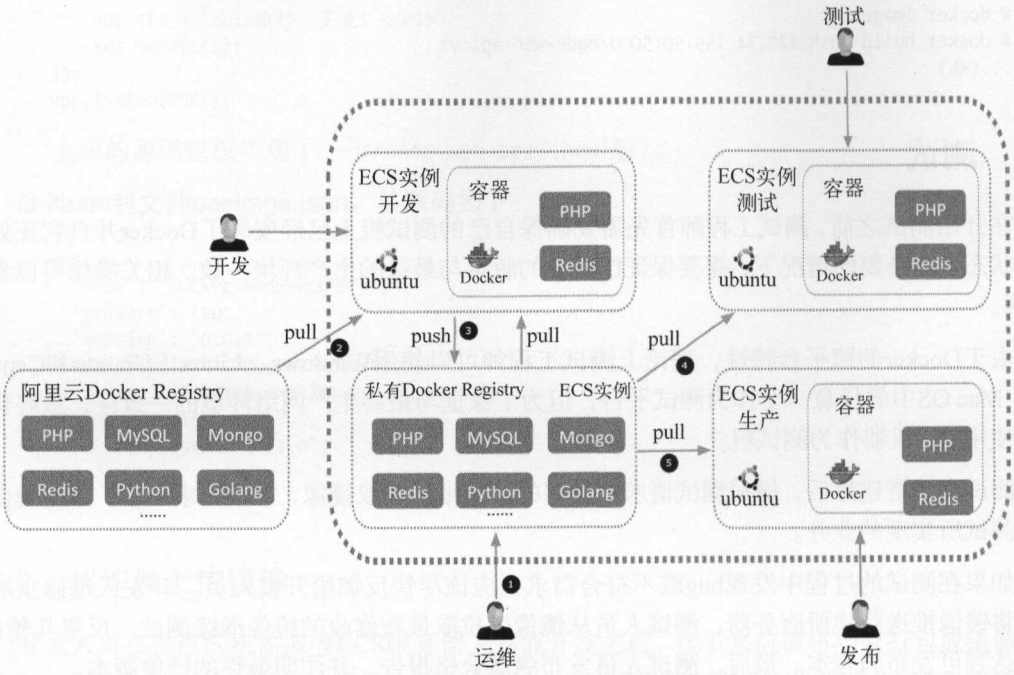


图8-6　ECS Docker实践指示（摘自阿里云官网）

第9章 在阿里云上部署WordPress

WordPress是一个基于PHP语言和MySQL数据库实现的免费开源的博客系统及内容管理系统。它拥有强大的插件系统架构和主题模板系统，不但可以用做美观、易用的个人博客网站，还能用来构建功能强大的网络信息发布平台。全世界Alexa排行前100万的网站中，就有超过16.7%的网站使用WordPress。它已经成为目前因特网上最流行的博客系统。

当我们准备以WordPress来搭建一个个人博客网站时，通常需要安装和准备以下几个模块：Apache 2、PHP 5、MySQL、WordPress源码包。依赖软件和源码包准备就绪后，还需完成一系列繁杂的配置操作，比如WordPress要连接的数据库、Apache的PHP加载模块、Apache URL重写支持等。除了安装复杂外，它在隔离性、安全性、资源限制和后期管理等方面也存在诸多挑战，而这些问题在Docker容器技术出现后有了明显改观。

本章将介绍如何在阿里云上使用Docker技术来部署个人博客网站。相比传统的部署方式，我们将体验到使用Docker技术的简单与快捷。本章主要内容包括：

- 初始化阿里云环境，主要是主机环境的配置；
- 部署MySQL容器；
- 部署WordPress容器。

9.1 初始化阿里云 Docker 环境

首先，登录到阿里云的官网（http://www.aliyun.com/），购买ECS主机。图9-1展示了ECS选项配置界面。用作个人博客站点的网站，我们选用图9-1展示的配置即可，性价比相对较高。因为我们将同时运行两个容器服务（MySQL和WordPress），为了保证性能，内存需要选择1GB或以上。另外，某些Docker镜像以默认配置在低内存状态下运行会出现异常，如MySQL。在选择系统镜像时，我们直接从镜像市场中选择Docker运行环境（Ubuntu 64位）镜像，这样可省去Docker繁杂的安装配置过程。

mysql和wordpress镜像的官方地址分别是https://registry.hub.docker.com/_/mysql/和https://registry.hub.docker.com/u/library/wordpress/。

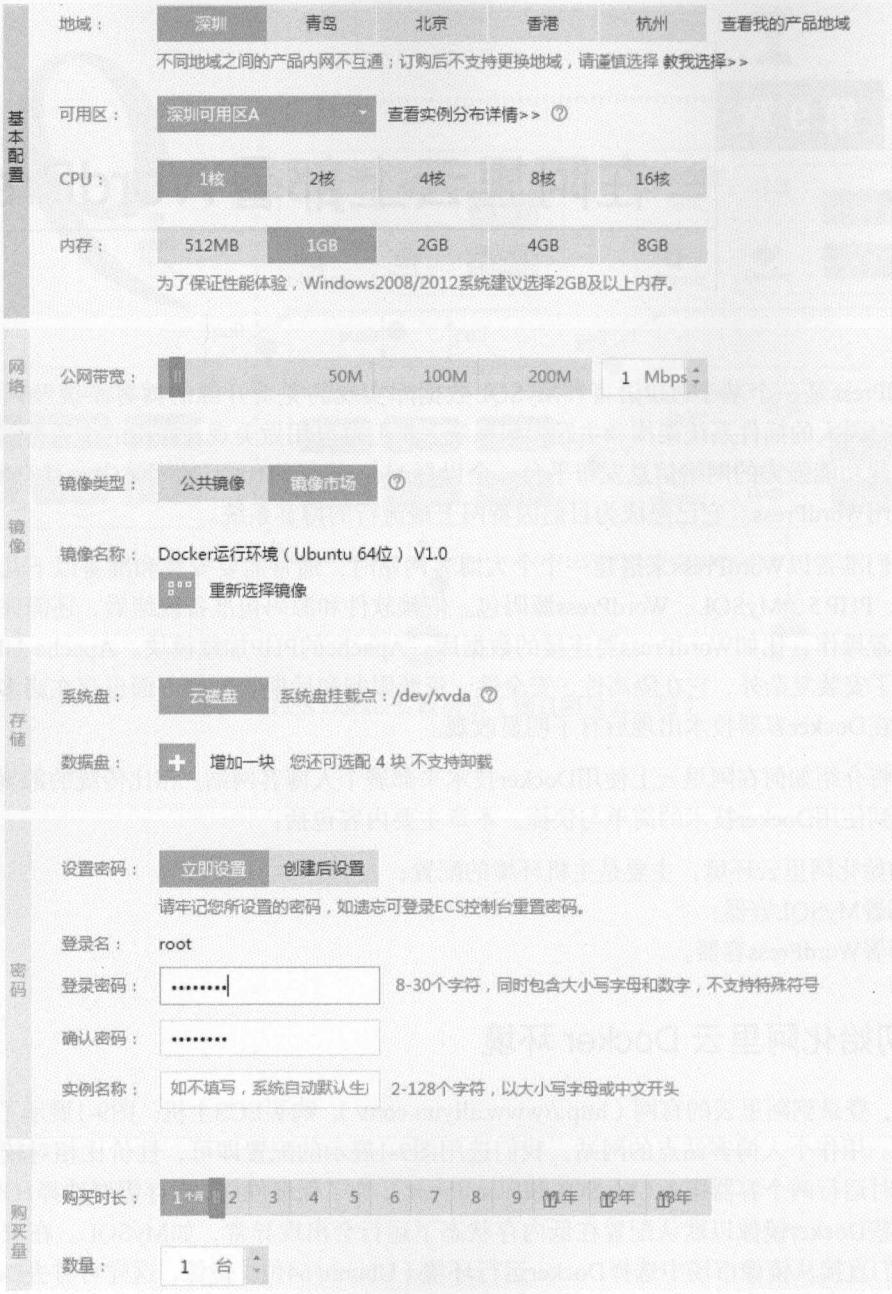


图9-1　购买ECS选项界面

我们可以直接从阿里云提供的注册服务器拉取mysql和wordpress镜像。因为使用的是阿里云内部连接，这比到国外的Docker Hub上拉取镜像要快很多。以下代码展示了具体的操作过程：

```
root@iZ94nflqok7Z:~# docker pull registry.mirrors.aliyuncs.com/library/mysql:latest
Pulling repository registry.mirrors.aliyuncs.com/library/mysql
...(略)
Status: Downloaded newer image for registry.mirrors.aliyuncs.com/library/mysql:latest
root@iZ94nflqok7Z:~# docker pull registry.mirrors.aliyuncs.com/library/wordpress:latest
Pulling repository registry.mirrors.aliyuncs.com/library/wordpress
...(略)
Status: Downloaded newer image for registry.mirrors.aliyuncs.com/library/wordpress:latest
```

9.2 部署 MySQL 容器

MySQL是全世界最流行的开源的关系型数据库管理系统，特别适用于网络应用程序，常与Linux、Apache、PHP搭配成为目前最流行的Web后台开发技术栈之一，合并称为LAMP。而WordPress也正是典型的基于LAMP技术栈开发的Web系统。我们仅需要下面的一条命令就能启动我们博客系统的MySQL数据库服务：

```
# docker run --name xxh-blog-db -e MYSQL_ROOT_PASSWORD=mysqlrootpassword -d
registry.mirrors.aliyuncs.com/library/mysql
```

需要注意的是，镜像名指定的是阿里云所提供的镜像。MySQL服务默认是在3306端口监听服务。因为并不需要跨主机连接MySQL，所以也就没必要将该端口暴露到主机，这样同时能提高数据库的安全性。在下一节中，我们会讲到使用--link参数将WordPress直接连接到数据库。

使用环境变量来配置新启动的容器环境是初始化容器最常见的一种手段。下面介绍MySQL所支持的几个环境变量。

- MYSQL_ROOT_PASSWORD。这个环境变量是必选的，用来指定MySQL容器服务的根用户（root）密码。上面的例子中所指定的密码是mysqlrootpassword。
- MYSQL_USER和MYSQL_PASSWORD。这两个环境变量是可选的，用来额外添加一个MySQL用户并指定其密码。这两个变量必须同时指定，缺少其中任何一个，另一个参数也不会产生效果。我们不能使用这个环境变量来添加root用户，因为root用户是默认存在的，并且root用户的密码是通过MYSQL_ROOT_PASSWORD来指定的。
- MYSQL_DATABASE。这个环境变量是可选的，用来额外创建一个数据库，其参数为数据库名称。如果同时指定了MYSQL_USER和MYSQL_PASSWORD，那么MYSQL_USER用户将拥有访问此数据库的所有权限。

9.3 部署 WordPress 容器

数据库服务启动后，就可以启动WordPress容器了。使用下面的命令启动：

```
# docker run --name xxh-blog --link xxh-blog-db:mysql -p 80:80 -d
registry.mirrors.aliyuncs.com/library/wordpress
```

我们使用link标记将上一节启动的xxh-blog-db数据库容器以约定的固定别名mysql导入到WordPress容器内。因为我们的博客最终是要对外网提供访问的，所以还要通过-p 80:80标记将WordPress容器默认监听的80 端口映射到主机的80端口上。在此，我们需要确保主机上的80端口没有被其他应用占用，否则将导致启动失败。容器启动成功后，在外网使用http://ecs-host-ip/地址就可以访问到我们的博客了。笔者测试时，使用的IP地址是120.24.159.50。但首次访问时，会被定向到WordPress安装页面，如图9-2所示，我们可以根据提示完成站点标题、用户名、密码、电子邮箱的填写。

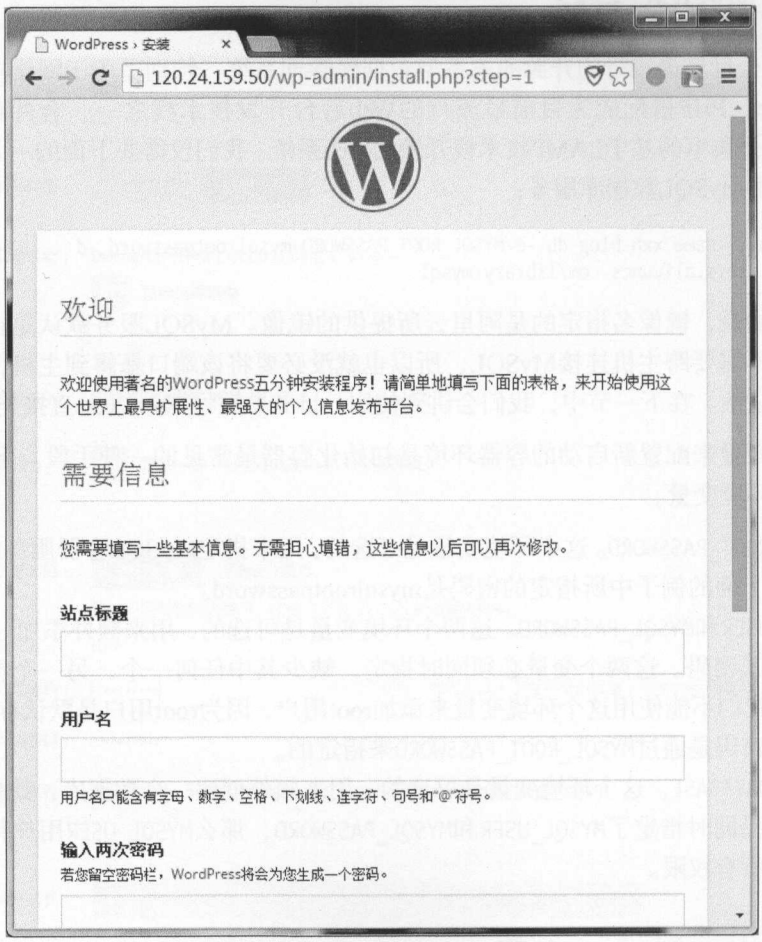


图9-2　WordPress安装界面

安装完成后，再次通过http://ecs-host-ip/地址就能直接进入博客主页了。使用安装时填写的用户名和密码登录到博客的仪表盘页面，在这里可以编写和发布博客并且对WordPress的方方面面进行配置，如图9-3所示。

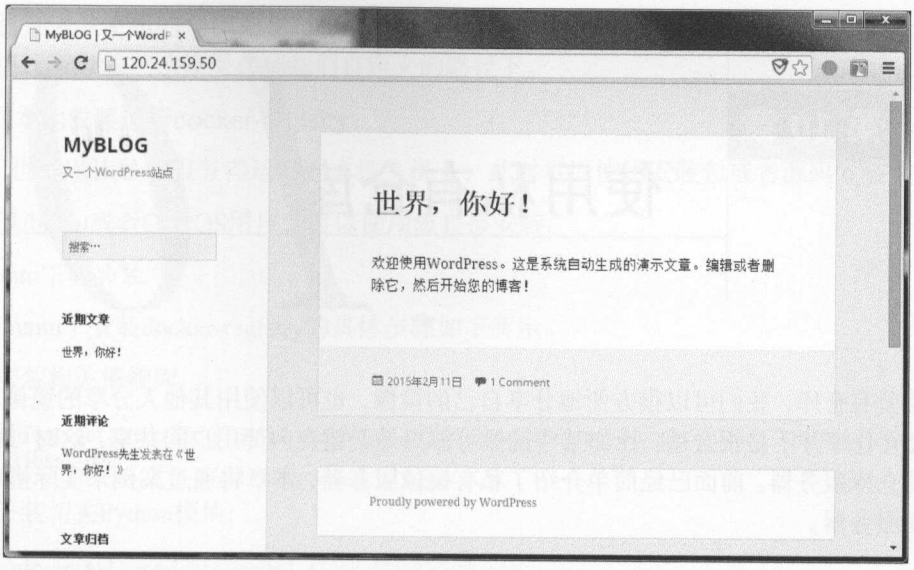


图9-3 博客主页面

通过IP地址访问终究不是很方便，用户可以到万网http://www.net.cn/购买适合自己的域名，并到阿里云代备案管理系统https://beian.gein.cn/完成域名备案，最后将域名绑定到你的ECS服务器的IP地址，这样就能通过域名访问博客了。这方面的详细操作不在本书的讨论范围内，读者可以到各自官网获取帮助。

同样，WordPress也支持使用额外的环境变量来定制。

- WORDPRESS_DB_USER。连接MySQL数据库服务的用户名。如果不指定，则使用默认root作为用户。
- WORDPRESS_DB_PASSWORD。连接MySQL数据库服务的密码，如果不指定，就使用连接到的MySQL容器中的环境变量MYSQL_ROOT_PASSWORD作为密码。
- WORDPRESS_DB_NAME。WordPress要使用的数据库名称。如果不指定，则使用默认值wordpress。如果目标数据库服务中不存在这个数据库，那么容器在启动时会自动创建该数据库，前提是WORDPRESS_DB_USER用户拥有创建数据库的权限。
- WORDPRESS_AUTH_KEY、WORDPRESS_SECURE_AUTH_KEY、WORDPRESS_LOGGED_IN_KEY、WORDPRESS_NONCE_KEY、WORDPRESS_AUTH_SALT、WORDPRESS_SECURE_AUTH_SALT、WORDPRESS_LOGGED_IN_SALT、WORDPRESS_NONCE_SALT。这一系列环境变量用来配置WordPress的安全性，容器默认使用唯一的随机SHA1值。我们使用默认值即可。

第 10 章 使用私有仓库

使用公有仓库，我们可以很方便地分享自己的镜像，也可以使用其他人分享的镜像。但是有时候，公有仓库并不是很合适，比如某些镜像可能只是希望在内部用户间共享，这时可以搭建一个私有的仓库服务器。前面已经简单介绍了私有镜像服务器，本章将通过案例来实际搭建一个本地的仓库服务器。

10.1 使用 docker-registry

docker-registry是一个基于Python的开源项目，为我们提供了搭建私有镜像服务器的功能。我们既可以在真正的主机上运行docker-registry，也可以在容器中运行它。官方仓库中已经提供了docker-registry的镜像，本节将介绍这两种搭建方式。

1. 使用容器运行docker-registry

首先，使用docker run命令获取并运行官方的镜像：

```
$ sudo docker run -d -p 5000:5000 registry
```

通过上面的命令，我们的私有服务器就以默认参数运行了。我们也可以配合使用-e和-v参数来改变服务器的运行参数。这里比较重要的有两个参数：一是配置文件的路径，二是仓库的路径。下面简要介绍一下这两个参数。

- **配置文件的路径。**这通过类似下面的命令来改变：

```
$ sudo docker run -d -p 5000:5000 -v /home/share/registry-conf:/root/registry-conf -e
DOCKER_REGISTRY_CONFIG=/root/registry-conf/config.yml registry
```

在本地目录/home/share/registry-conf下存放着要使用的配置文件，通过-v参数把它映射到容器的/root.registry-conf目录，使用-e用环境变量的方式指定/root/registry-conf/config.yml为程序的配置文件。

- **配置仓库路径。**这通过类似下面的命令来改变：

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

官方镜像使用的仓库路径是/tmp/registry/，我们通过把本地路径映射到/tmp/registry/，可以让服务器把镜像保存到我们自定义的路径下。

2. 在本地安装运行docker-registry

我们也可以让私有服务器运行在本地主机上，此时可以使用安装包或者源码安装。

对于Ubuntu或者CentOS用户，可以使用源直接安装。

Ubuntu下的安装

在Ubuntu下安装docker-registry的具体步骤如下所示。

(1) 安装相关依赖库:

```
$ sudo apt-get install -y build-essential python-dev libevent-dev python-pip liblzma-dev
swig libssl-dev
```

(2) 安装相关Python模块:

```
sudo pip install gunicorn pyyaml flask flask-cors rsa
```

(3) 安装docker-registry:

```
sudo pip install docker-registry
```

CentOS下的安装

在CentOS下安装docker-registry的具体步骤如下所示。

(1) 安装相关依赖库：

```
sudo yum install -y python-devel libevent-devel python-pip gcc xz-devel
```

(2) 安装相关Python模块：

```
sudo pip install gunicorn pyyaml flask flask-cors rsa gevent
```

(3) 安装docker-registry：

```
sudo python-pip install docker-registry
```

使用源码安装

使用源码安装docker-registry的具体步骤如下所示。

(1) 安装相关的依赖库：

```
sudo apt-get install -y build-essential python-dev libevent-dev python-pip libssl-dev
liblzma-dev libffi-dev
```

(2) 从docker-registry项目下载源码：

```
git clone https://github.com/docker/docker-registry.git
```

默认情况下只有样例配置文件，正式的配置文件要我们自己生成。当然，我们也可以简单地修改样例来生成配置：

```
cd docker-registry
cp config/config_sample.yml config/config.yaml
```

修改如下部分，配置仓库存储路径：

```
local: &local
    <<: *common
    storage: local
    storage_path: _env:STORAGE_PATH:/tmp/registry
```

(3) 安装docker-registry：

```
sudo python setup.py install
```

通过源方式安装的docker-registry默认也没有配置，也要我们自己生成，按上面的来就好了。配置文件的路径在/usr/local/lib/python2.7/dist-oackages/docker_registry/config/config.yaml。

3. 启动服务器

通过下面的命令，可以启动服务器：

```
sudo gunicorn --access-logfile /var/log/docker-registry/access.log --error-logfile
/var/log/docker-registry/server.log -k gevent --max-requests 100 --graceful-timeout 3600 -t 3600 -b
127.0.0.1:5000 -w 1 docker_registry.wsgi:application
```

用HTTP方式访问本地的5000端口，如果看到类似下面的输出，就说明服务器正常运行了：

```
$ curl 127.0.0.1:5000
"\"docker-registry server\""
```

可以看到，前面的命令有些长，每次输入一大串很不方便，这时可以通过服务脚本来管理私有服务器。下面以Ubuntu为例进行介绍。

(1) 按下面的内容创建并编辑/etc/init/docker-registry.conf文件：

```
#sudo cat /etc/init/docker-registry.conf
description "Docker Registry"
start on runlevel [2345]
stop on runlevel [016]
respawn
respawn limit 10 5
script
sudo gunicorn --access-logfile /var/log/docker-registry/access.log --error-logfile
/var/log/docker-registry/server.log -k gevent --max-requests 100 --graceful-timeout 3600 -t 3600 -b
127.0.0.1:15000 -w 8 docker_registry.wsgi:application
```

```
end script
#service docker-registry start
```

(2) 在命令行执行service docker-registry start，registry服务会在本地的15000端口监听。

10.2 用户认证

一般情况下，只有授权的用户才可以访问和使用我们的私有服务器。Docker Hub是通过注册索引服务器（index）来实现的。由于index服务器还没有完善的开源实现，所以这里基于Nginx代理的用户管理实现用户认证。

1. 安装并配置Nginx代理

上一节中，我们让registry服务监听127.0.0.1:15000端口，这样的话只有本机才能通过15000端口访问到服务器，其他主机是无法访问的。

为了让其他主机访问服务器，我们可以使用Nginx代理，让Nginx开放对外的5000端口，当外部的请求到达5000端口时，再由Nginx把请求转发到本地的15000端口。

首先，安装Nginx：

```
sudo apt-get install nginx
```

在/etc/nginx/sites-available/下创建站点配置文件docker-registry.conf，实现我们的代理目的。

下面为配置文件的具体内容：

```
upstream docker-registry {
    server localhost:15000;
}
#代理服务器，监听5000端口
server {
    listen 5000;
    server_name private-registry-server.com;
    proxy_set_header Host          $http_host;
    proxy_set_header X-Real_IP     $remote_addr;  #传递真实的客户端IP
    client_max_body_size 0;  #disable any limits to avoid HTTP 413 for large image uploads
    chunked_transfer_encoding on;
    location / {
        #配置转发对于/的访问请求到registry服务
        proxy_pass http://docker-registry;
    }
    location / {
        #配置转发对于/的访问请求到registry服务
        proxy_pass http://docker-registry;
    }
    location / {
        #配置转发对于/的访问请求到registry服务
        proxy_pass http://docker-registry;
    }
}
```

上面的配置文件已经建好了，但是还没有生效，把它用软连接放到/etc/nginx/sites-enabled/目录下，Nginx就可以启动代理了。最后，重启Nginx使配置生效：

```
sudo ln -s /etc/nginx/sites-available/docker-registry.conf
/etc/nginx/sites-enabled/docker-registry.conf
service nginx restart
```

通过访问本地的5000端口来测试代理服务是否生效，这里尝试上传一个本地镜像：

```
sudo docker tag ubuntu:14.04 127.0.0.1:5000/ubuntu:latest
sudo docker push 127.0.0.1:5000/ubuntu:latest
```

2. 添加用户认证功能

添加用户认证功能的具体步骤如下所示。

(1) 在前面的代理站点配置文件的相应部分添加下面粗体显示的代码：

```
...
location / {
    #配置认证文件
    auth_basic            "Please Input username&password";
    auth_basic_user_file    docker-registry-htpasswd;
    #配置转发对于/的访问请求到registry服务
    proxy_pass http://docker-registry;
}
...
```

在上述代码中，auth_basic行告诉Nginx启用认证服务，auth_basic_user_file行则指定认证信息存放的文件（在/etc/nginx/目录下）。

(2) 重启Nginx服务：

```
$ sudo service nginx restart
```

现在通过浏览器访问本地的代理服务http://127.0.0.1:5000，会弹出让我们输入用户名和密码的对话框。当然，现在还没有添加任何用户，所以无法访问。添加用户要用到htpasswd工具，通过以下命令来安装htpasswd工具：

```
sudo apt-get install apache2-utils -y
```

(3) 现在添加一个用户试试：

```
$ sudo htpasswd -c  /etc/nginx/docker-registry-htpasswd test
New password:
Re-type new password:
Adding password for user test
```

上面的命令添加了test用户，接下来提示我们输入密码并确认（输入是看不到的，不要担心）。最后，会提示我们用户添加成功了。

(4) 使用刚才的用户名和密码测试一下：

```
$ curl -u test:pass 127.0.0.1:5000/v1/search
{"num_results": 1, "query": "", "results": [{"description": "",
    "name": "library/ubuntu"}]}
```

可以看到，成功通过了认证，并且可以查询到前面添加的镜像。

第 11 章 使用Docker部署Hadoop集群

Hadoop是一款开源的分布式计算框架，它的出现极大地促进了云计算平台的发展。在Hadoop出现之前，要搭建一个分布式网络并不是件容易的事，而且进行分布式编程也相当费劲。Hadoop通过它的MapReduce计算模型和HDFS存储模型，极大地简化了计算，隔离了存储细节，使得编写分布式应用变得格外简单。正因为此，Hadoop也成了云计算领域的宠儿，时至今日扔热度不减。Hadoop和Docker作为云计算领域的两大热门技术，无视它们之间的关系等于掩耳盗铃，所以在本章中，我们将说明它们两个是如何走到一起的。本章的主要内容有：

- Hadoop简介；
- 构建hadoop镜像。通过Dockerfile来构建hadoop镜像；
- 构建Hadoop集群。通过ambari镜像来部署和管理Hadoop集群。

11.1 Hadoop 简介

Hadoop是由Apache软件基金会所开发的分布式系统基础架构，它可以让开发者不了解分布式环境的底层细节，就能够快速开发出高效、健壮的分布式程序。该项目最重要的两个核心概念是MapReduce和HDFS，前者负责处理海量数据，后者则为海量数据提供可靠的存储。MapReduce分为两个动作——Map和Reduce，分别实现任务的分解和结果的汇总。HDFS（Hadoop Distributed File System）分布式文件系统具有高度容错的优点，它本意就是用来将数据部署在大规模的低廉硬件上，所以极大地降低了分布式环境的搭建门槛和成本，也是硬件量化和流式分配的基础。Hadoop主要具有以下优点。

- **可扩展性**：不论是计算还是存储，Hadoop都具备良好的可扩展性，这也是Hadoop的设计根本。
- **经济**：Hadoop的运行环境是普通的PC集群上，硬件要求低。
- **可靠**：HDFS冗余备份、数据恢复机制，以及MapReduce的任务监控，都保证了分布式处理的可靠性。
- **高效**：Hadoop尽量减少数据的移动，采用计算跟随数据的设计，减少了数据传输带来的时延，从而在处理海量数据上具有很高的效率。

Hadoop主要应用在海量数据分析、分布式存储、Web搜索引擎等应用领域。关于Hadoop更多的知识，我们在此不再展开。接下来，我们主要说明如何构建Hadoop镜像以及如何通过Docker来部署Hadoop集群。

11.2 构建 Hadoop 镜像

获得Hadoop镜像有两种方式，一种是直接从Docker Hub中拉取现有的Hadoop镜像，另一种是通过Dockerfile来构建。

直接拉取现有的Hadoop镜像的代码如下：

```
$ sudo docker pull sequenceiq/hadoop-docker:2.6.0
```

下面我们重点介绍如何通过Dockerfile来构建Hadoop镜像，具体步骤如下所示。

(1) 下载文件。通过wget命令或者网络浏览器获取Dockerfile以及相关配置文件：

```
wget https://github.com/minimicall/hadoop-docker/archive/master.zip
```

(2) 解压文件：

```
$ unzip -o -d ./hadoop master.zip
```

进入解压后的目录，可以看到如下文件列表：

```
$ ls
bootstrap.sh            hadoop          mapred-site.xml  ssh_config
core-site.xml.template  hdfs-site.xml   master.zip       yarn-site.xml
Dockerfile              LICENSE         README.md
```

可以发现，除了Dockerfile外，还有一些构建镜像中需要用到的配置文件。

(3) 查看Dockerfile文件的内容。

下面我们简要说明Dockerfile的内容，其中以注释的方式加以说明：

```
# Creates pseudo distributed hadoop 2.6.0
#
# docker build -t sequenceiq/hadoop .
#基础镜像
FROM sequenceiq/pam:centos-6.5
MAINTAINER SequenceIQ
#切换到root用户
USER root
#安装必要的开发工具
RUN yum install -y curl which tar sudo openssh-server openssh-clients rsync
# update libselinux. see https://github.com/sequenceiq/hadoop-docker/issues/14
RUN yum update -y libselinux
#配置无需密码的SSH连接
RUN ssh-keygen -q -N "" -t dsa -f /etc/ssh/ssh_host_dsa_key
RUN ssh-keygen -q -N "" -t rsa -f /etc/ssh/ssh_host_rsa_key
```

```
RUN ssh-keygen -q -N "" -t rsa -f /root/.ssh/id_rsa
RUN cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys
#Java JDK的获取
RUN curl -LO 'http://download.oracle.com/otn-pub/java/jdk/7u51-b13/jdk-7u51-linux-x64.rpm' -H 'Cookie:
    oraclelicense=accept-securebackup-cookie'
#Java JDK的安装
RUN rpm -i jdk-7u51-linux-x64.rpm
#删除rpm包
RUN rm jdk-7u51-linux-x64.rpm
#配置环境变量JAVA_HOME和PATH
ENV JAVA_HOME /usr/java/default
ENV PATH $PATH:$JAVA_HOME/bin
#Hadoop的下载和解压
RUN curl -s http://www.eu.apache.org/dist/hadoop/common/hadoop-2.6.0/hadoop-2.6.0.tar.gz | tar -xz -C
    /usr/local/
#制作链接文件
RUN cd /usr/local && ln -s ./hadoop-2.6.0 hadoop
#设置环境变量
ENV HADOOP_PREFIX /usr/local/hadoop
ENV HADOOP_COMMON_HOME /usr/local/hadoop
ENV HADOOP_HDFS_HOME /usr/local/hadoop
ENV HADOOP_MAPRED_HOME /usr/local/hadoop
ENV HADOOP_YARN_HOME /usr/local/hadoop
ENV HADOOP_CONF_DIR /usr/local/hadoop/etc/hadoop
ENV YARN_CONF_DIR $HADOOP_PREFIX/etc/hadoop
#将JAVA及Hadoop相关环境变量配置进hadoop-en.sh配置文件中
RUN sed -i '/^export JAVA_HOME/ s:.*:export JAVA_HOME=/usr/java/default\nexport
HADOOP_PREFIX=/usr/local/hadoop\nexport HADOOP_HOME=/usr/local/hadoop\n:'
$HADOOP_PREFIX/etc/hadoop/hadoop-env.sh
RUN sed -i '/^export HADOOP_CONF_DIR/ s:.*:export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop/:'
    $HADOOP_PREFIX/etc/hadoop/hadoop-env.sh
#RUN . $HADOOP_PREFIX/etc/hadoop/hadoop-env.sh
RUN mkdir $HADOOP_PREFIX/input
RUN cp $HADOOP_PREFIX/etc/hadoop/*.xml $HADOOP_PREFIX/input
#伪分布式环境配置，需要将本地同一目录下的配置文件复制进容器内部
# pseudo distributed
ADD core-site.xml.template $HADOOP_PREFIX/etc/hadoop/core-site.xml.template
RUN sed s/HOSTNAME/localhost/ /usr/local/hadoop/etc/hadoop/core-site.xml.template >
/usr/local/hadoop/etc/hadoop/core-site.xml
#添加hdfs、mapred、yarn站点配置文件
ADD hdfs-site.xml $HADOOP_PREFIX/etc/hadoop/hdfs-site.xml
ADD mapred-site.xml $HADOOP_PREFIX/etc/hadoop/mapred-site.xml
ADD yarn-site.xml $HADOOP_PREFIX/etc/hadoop/yarn-site.xml
#格式化namenode
RUN $HADOOP_PREFIX/bin/hdfs namenode -format
# fixing the libhadoop.so like a boss
RUN rm  /usr/local/hadoop/lib/native/*
RUN curl -Ls http://dl.bintray.com/sequenceiq/sequenceiq-bin/hadoop-native-64-2.6.0.tar | tar -x -C
    /usr/local/hadoop/lib/native/
#SSH配置
ADD ssh_config /root/.ssh/config
RUN chmod 600 /root/.ssh/config
RUN chown root:root /root/.ssh/config
# # installing supervisord
```

```
# RUN yum install -y python-setuptools
# RUN easy_install pip
# RUN curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py -o - | python
# RUN pip install supervisor
#
# ADD supervisord.conf /etc/supervisord.conf
ADD bootstrap.sh /etc/bootstrap.sh
RUN chown root:root /etc/bootstrap.sh
RUN chmod 700 /etc/bootstrap.sh
ENV BOOTSTRAP /etc/bootstrap.sh
# workingaround docker.io build error
RUN ls -la /usr/local/hadoop/etc/hadoop/*-env.sh
RUN chmod +x /usr/local/hadoop/etc/hadoop/*-env.sh
RUN ls -la /usr/local/hadoop/etc/hadoop/*-env.sh
# fix the 254 error code
RUN sed  -i "/^[^#]*UsePAM/ s/.*/#&/"  /etc/ssh/sshd_config
RUN echo "UsePAM no" >> /etc/ssh/sshd_config
RUN echo "Port 2122" >> /etc/ssh/sshd_config
#启动相关服务
RUN service sshd start && $HADOOP_PREFIX/etc/hadoop/hadoop-env.sh && $HADOOP_PREFIX/sbin/start-dfs.sh
    && $HADOOP_PREFIX/bin/hdfs dfs -mkdir -p /user/root
RUN service sshd start && $HADOOP_PREFIX/etc/hadoop/hadoop-env.sh && $HADOOP_PREFIX/sbin/start-dfs.sh
    && $HADOOP_PREFIX/bin/hdfs dfs -put $HADOOP_PREFIX/etc/hadoop/ input
#入口命令
CMD ["/etc/bootstrap.sh", "-d"]
#对外暴露的网络端口
EXPOSE 50020 50090 50070 50010 50075 8031 8032 8033 8040 8042 49707 22 8088 8030
```

(4) 构建Hadoop镜像。

在Dockerfile所在目录中执行docker build命令来构建Hadoop镜像：

```
$ docker build  -t sequenceiq/hadoop-docker:2.6.0 .
...
Step 51 : EXPOSE 50020 50090 50070 50010 50075 8031 8032 8033 8040 8042 49707 22 8088 8030
 ---> Running in 83a985d1b344
 ---> 671d9ac19702
Removing intermediate container 83a985d1b344
Successfully built 671d9ac19702
```

使用Dockerfile构建Hadoop镜像时，每一条命令都将新建一层，当构建完新的一层时，旧的中间镜像就会被删除，最终看到Successfully built ...就表明构建成功。

构建完毕之后，通过docker images命令来查看本地镜像：

```
micall@micall-ThinkPad:~/docker/hadoop-docker-master$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
minimicall/hadoop-docker  2.6.0               671d9ac19702        7 hours ago         1.636 GB
sequenceiq/hadoop-docker  2.6.0               d828dda7ad02        4 weeks ago         1.624 GB
```

可以看到，我们直接拉取的镜像sequenceiq/hadoop-docker:2.6.0和通过Dockerfile构建的镜像minimicall/hadoop-docker:2.6.0。

有了镜像之后，我们就可以通过docker run命令来启动Hadoop容器了：

```
docker run -it sequenceiq/hadoop-docker:2.6.0 /etc/bootstrap.sh -bash
/
Starting sshd:                                             [  OK  ]
Starting namenodes on [91f69f9e5ab3]
Starting secondary namenodes [0.0.0.0]
starting yarn daemons
...
bash-4.1#
```

可以看到，Hadoop容器先后启动了sshd、namenodes、secondary namenodes和yarn等服务组件。最后，进入到了Hadoop容器的命令窗口。下面运行Hadoop自带的一个示例来测试Hadoop程序：

```
bash-4.1# cd /usr/local/hadoop
bash-4.1# bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.6.0.jar grep input output
'dfs[a-z.]+'
15/02/12 21:13:18 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
15/02/12 21:13:20 WARN mapreduce.JobSubmitter: No job jar file set.  User classes may not be found.
See Job or Job#setJar(String).
15/02/12 21:13:20 INFO input.FileInputFormat: Total input paths to process : 31
15/02/12 21:13:21 INFO mapreduce.JobSubmitter: number of splits:31
15/02/12 21:13:31 INFO mapreduce.Job:  map 0% reduce 0%
...
15/02/12 21:14:02 INFO mapreduce.Job:  map 19% reduce 0%
...
15/02/12 21:15:16 INFO mapreduce.Job:  map 100% reduce 100%
...
```

程序成功执行了，该程序主要实现一个以dfs为前缀的词频统计工作，其执行结果为：

```
# bin/hdfs dfs -cat output/*
6 dfs.audit.logger
4 dfs.class
3 dfs.server.namenode.
2 dfs.period
2 dfs.audit.log.maxfilesize
2 dfs.audit.log.maxbackupindex
1 dfsmetrics.log
1 dfsadmin
1 dfs.servers
1 dfs.replication
1 dfs.file
```

通过该容器，就可以进行Hadoop程序的开发和运行了。然而Hadoop一般在集群上才更能体现其价值，而该容器只是一个Hadoop节点。接下来，我们介绍一下如何通过Docker来部署Hadoop集群。

11.3　构建 Hadoop 集群

在这一节中，我们介绍一款管理Hadoop集群的工具Ambari，以及如何通过Docker来部署Hadoop集群。

11.3.1 Ambari 简介

Apache Ambari是一款致力于简化Hadoop集群的安装、开发集成、管理和监控的开源软件。它对外提供RESTful形式的API，以供外部更好地管理Hadoop集群。通过Ambari工具，系统管理员可以进行如下操作。

- **安装Hadoop集群**：Ambari提供了在集群中构建Hadoop服务的步骤以及安装配置。
- **管理Hadoop集群**：Ambari提供了对Hadoop集群中所有节点的集中管理，包括启动、停止和重新配置等。
- **监控Hadoop集群**：Ambari提供了监控Hadoop集群健康状况和运行状态的控制面板，还对外提供监控接口，以便用户定制监控信息。

通过Ambari的RESTful API接口，Hadoop的安装、管理和监控等组件也特别容易持续集成。Ambari是Hadoop集群开发的一把利器，本节中我们将通过Ambari的Docker镜像来构建Hadoop集群环境。

Ambari框架采用的是C/S模式，其架构如图11-1所示。服务端是ambari-server，客户端是ambari-agent。ambari-server主要管理部署在每个节点上的管理监控程序，对外提供RESTful API，其中一类API为ambari-web程序（ambari-web是Ambari提供的Web形式的管理工具）提供监控管理服务，另一类API与ambari-agent交互。ambari-server中的Master模块接受API和Agent接口的请求，完成集群的集中式管理监控逻辑，并将相关状态保存到PostgreSQL数据库中。ambari-agent分布在集群的每一个节点，它是无状态的，只负责所在节点的状态采集及维护工作。此外，Ambari还提供了ambari-shell，以shell的形式来管理集群。在ambari-shell内部，通过API和ambari-server进行通信。

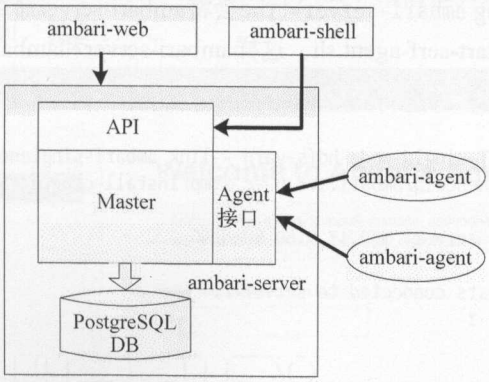


图11-1 Ambari的C/S架构

11.3.2 部署 Hadoop 集群

在说明必要的基础原理之后，下面开始Ambari的Docker之旅。

首先，我们可以在Docker Hub上直接获取Ambari的镜像：

```
$ docker pull sequenceiq/ambari:1.6.0
```

拉取成功之后，查看本地镜像：

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
sequenceiq/ambari       1.6.0               bec373d0401a        3 months ago        1.782 GB
```

这个镜像包含了ambari-server、ambari-agent和ambari-shell，你可以通过它们部署单点或者多点的Hadoop集群。

这里我们部署一个单点的Hadoop集群。对于多点集群，具体步骤也是一样的，只是涉及更多具体的Ambari配置，这里就不展开说明了。这里将会用到两个容器，它们都基于刚刚获取到的镜像。第一个容器运行ambari-server和ambari-agent，第二个容器则运行ambari-shell，用于通过命令的形式管理集群。

首先，启动运行ambari-server和ambari-agent的容器：

```
$ docker run -d -p 8080 -h 127.0.0.1 --name ambari-singlenode sequenceiq/ambari:1.6.0 --tag
ambari-server=true
123f5b8a3f36229357310e41bff15c6a8d871b08e7316c2bc211cbdf2fba36f4
micall@micall-ThinkPad:~$ docker ps
CONTAINER ID        IMAGE                      COMMAND                  CREATED
STATUS              PORTS                      NAMES
123f5b8a3f36        sequenceiq/ambari:1.6.0    "/usr/local/serf/bin   15 seconds ago
Up 13 seconds       7373/tcp, 7946/tcp, 0.0.0.0:49154->8080/tcp   ambari-singlenode
```

容器运行成功了。--tag ambari-server=true会将ambari-server命令加入到entrypoint中，而这个容器默认的entrypoint为start-serf-agent.sh，这样ambari-server和ambari-agent都运行起来了。

接着启动另外一个容器，即运行ambari-shell的容器：

```
$ docker run -e BLUEPRINT=single-node-hdfs-yarn --link ambari-singlenode:ambariserver -t --rm
--entrypoint /bin/sh sequenceiq/ambari:1.6.0 -c /tmp/install-cluster.sh
AMBARI_HOST=172.17.0.60
[DEBUG] waits for ambari server: 172.17.0.60 RUNNING ...
....
[DEBUG] waits until 1 hosts connected to server ...
[DEBUG] connected hosts: 1
                                                                                    _
   / \    _ __ ___  | |__    __ _  _ __ (_)/ ___| | |__    ___ | || |
  / _ \  | '_ ` _ \ | '_ \  / _` || '__|| |\___ \ | '_ \  / _ \| || |
 / ___ \ | | | | | || |_) || (_| || |   | | ___) || | | ||  __/| || |
/_/   \_\|_| |_| |_||_.__/  \__,_||_|   |_||____/ |_| |_| \___||_||_|


Welcome to Ambari Shell. For command and param completion press TAB, for assistance type 'hint'.
ambari-shell>blueprint defaults
Default blueprints added
```

```
ambari-shell>cluster build --blueprint single-node-hdfs-yarn
...
CLUSTER:single-node-hdfs-yarn>
```

启动容器后，就进入到ambari-shell命令界面中了。关于ambari-shell的用法，这里就不再展开说明了，读者可以参考相关资料进行学习。

我们知道Ambari还提供了Web管理界面。从上面的`docker ps`命令知道，主机的49154端口和第一个容器的8080端口进行映射了。我们可以通过浏览器来访问本地的49154端口，得到如图11-2所示的登录界面。

图11-2 Ambari的Web管理登录页面

默认的用户名和密码为admin/admin。登录之后，由于当前没有建立集群，所以会自动进入安装Hadoop集群的向导页面，见图11-3。

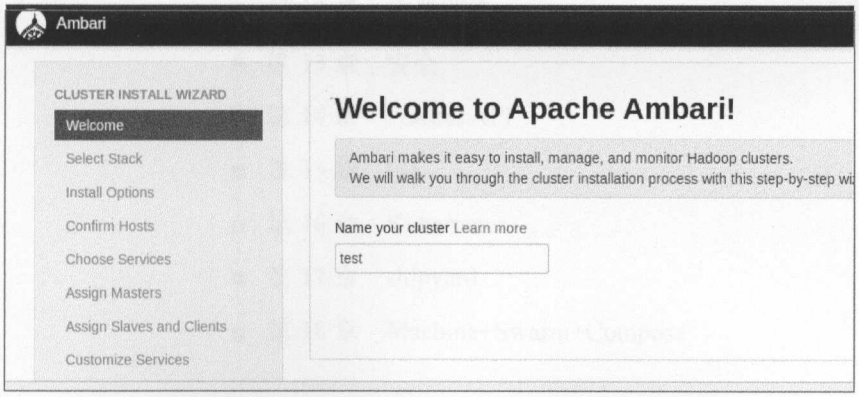


图11-3 Ambari的Hadoop集群安装向导页面

这里我们就不一步步说明如何安装Hadoop集群了。有了集群之后，就会出现如图11-4所示的管理面板。

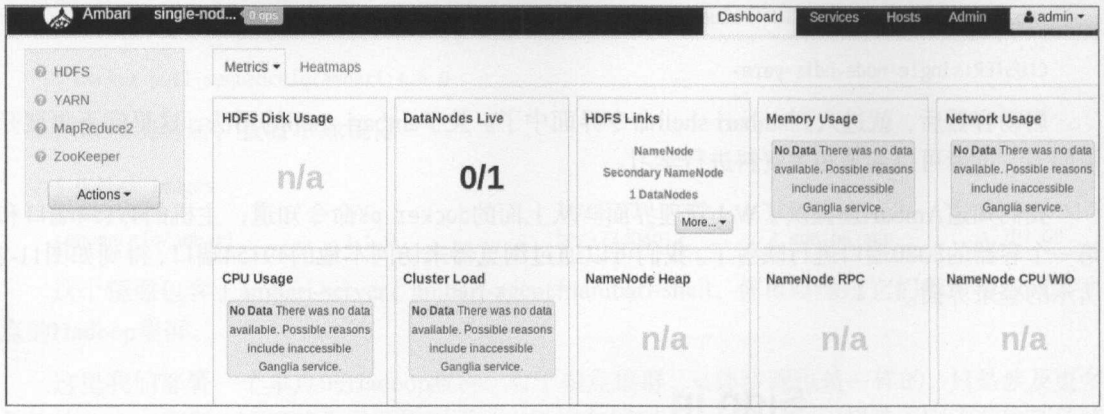


图11-4　Ambari的管理面板

关于使用Docker来部署Hadoop集群的介绍就到这里。若想了解更多相关信息，读者可以自行深入了解Ambari的用法。

Part 3

第三篇

高级篇：高级话题、API、工具及集群管理

本篇内容

- 第 12 章　容器网络
- 第 13 章　安全
- 第 14 章　Docker API
- 第 15 章　Fig
- 第 16 章　Kubernetes
- 第 17 章　shipyard
- 第 18 章　Machine+Swarm+Compose

第 12 章 容器网络 12

Docker最初诞生于Linux操作系统，在部署分布式网络应用方面提供了强有力的支持。只有对容器网络的原理理解透彻之后，才可以自定义出我们所需要的灵活的网络架构。本章主要包含以下内容：

- 与Docker紧密相关的网络工具。
- Docker网络的原理基础。
- 容器网络的配置及原理。
- Docker网桥的配置及自定义。

12.1 容器网络的原理

通过手动模拟Docker引擎内部的操作，能够让读者看清容器网络内部的本质。

12.1.1 基础网络工具

在讲解容器网络的原理之前，我们有必要先介绍几个与容器网络紧密相关的系统网络工具，比如iptables、ip和brctl-util。通过这些工具，我们可以检测容器的网络状态，也可以自定义我们的容器网络。

1. iptables

系统网络工具netfilter/iptables是Docker容器网络虚拟化所依赖的重要工具之一，它是Linux系统下一套IP数据包的过滤系统。通过定义一些规则，可以过滤、修改、重定向、阻塞流经主机的网络包。这些规则根据目的组织到不同的表（table）中，在表中又根据其检测的点组织成规则链。实际上，表和链可以看作是netfilter的两个维度。iptables共有4个表，分别是filter、nat、mangle和raw。iptables是用来查看和修改这些规则及配置的命令行工具。

表filter定义了哪些数据包可以通过，哪些不可以通过，展示如下：

```
# iptables -t filter -L
Chain INPUT (policy ACCEPT)
```

```
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

表nat定义了地址转换规则，展示如下：

```
# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere            !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       anywhere

Chain DOCKER (2 references)
target     prot opt source               destination
```

表mangle用来根据规则修改数据报文，展示如下：

```
# iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
```

```
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
```

在iptables中，-t命令用来指定要操作的表。由于raw默认情况下为空，且本章并不涉及该表，故以上展示没有将其列出来。

由上面的表可以看到，iptables内置了5条标准规则链，这些规则链也称为勾子函数（hook function）。

- PREROUTING：路由前勾子。
- INPUT：数据包流入口勾子。
- FORWARD：转发勾子。
- OUTPUT：数据包出口勾子。
- POSTROUTING：路由后勾子。

除了5个标准链外，用户还可以自定义链，如上面的DOCKER规则链。任何一个数据包，只要流经本机网络，必将触发以上勾子函数中的一个或多个。Docker容器也正是通过添加和修改相关的规则链来达到控制网络连接的目的的。

2. ip

Linux下的ip命令与ifconfig、route命令类似，但功能更强大，并旨在取代后者。ifconfig和route命令虽是已经废弃了的命令，但在大部分Linux系统上还能继续使用。本章中，我们会使用更现代的ip命令来做演示。如果读者更熟悉老的命令，也可以使用老命令。下面演示一下ip命令的几个功能。更多强大的操作，读者可参阅ip命令手册。

- 为指定网络设备设置IP：

```
# ip addr add 192.168.1.12/24 dev wlan0
```

- 查看指定网络设备的网络地址：

```
# ip addr show wlan0
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
link/ether 48:5d:60:ed:5a:0d brd ff:ff:ff:ff:ff:ff
inet 172.16.9.18/24 brd 172.16.9.255 scope global wlan0
   valid_lft forever preferred_lft forever
inet6 fe80::4a5d:60ff:feed:5a0d/64 scope link
   valid_lft forever preferred_lft forever
```

- 列出路由列表：

```
# ip route show
default via 172.16.9.1 dev wlan0  proto static
172.16.9.0/24 dev wlan0  proto kernel  scope link  src 172.16.9.18  metric 9
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.42.1
```

3. brctl-util

网桥是连接两个局域网的一种存储/转发设备，它能将一个大的LAN分割为多个网段，或将两个以上的LAN互联为一个逻辑LAN。网桥根据MAC地址来转发数据帧，可以看作一个“低层的路由器”，不过路由器工作在网络层，根据网络地址（如IP地址）进行数据包转发。

brctl-util工具是用来配置网桥的，同时可以在主机上创建和管理虚拟网桥。Ubuntu系统默认不带这个工具，需要先使用下面的命令安装该工具：

```
# apt-get install bridge-utils
```

下面演示brctl-util的几个常规用法。

- 查看本机网桥，以及连接在网桥上的网络设备：

```
# brctl show
bridge name bridge id  STP enabled interfaces
docker0  8000.56847afe9799 no  vethb926edc
                               vethcff2cc8
```

- 创建虚拟网桥：

```
# brctl addbr bridge0
```

- 删除网桥：

```
# brctl delbr bridge0
```

- 给网桥接入设备：

```
# brctl addif bridge0 veth0
```

关于brctl-util工具的更多操作，读者可查阅brctl命令手册。

12.1.2 网络空间虚拟化

在第1章中，我们已经了解了Docker容器的底层虚拟化技术，包括文件系统虚拟化、进程空间虚拟化、用户虚拟化和网络虚拟化等，其中网络虚拟化能够让一个容器处于与主机和其他容器完全独立的网络环境中，容器拥有自己私有的虚拟网卡、路由表及IP地址。当然，我们也可以让容器与主机或者其他容器共享同一个网络环境。

Docker网络虚拟化是基于Linux下的网络命名空间（net namespace）实现的。使用ip netns命令，我们同样可以创建自己的虚拟网络环境。下面的ip netns命令创建了一个名为net0的虚拟网络环境：

```
# ip netns add net0
# ip netns list
net0
```

这里需要说明的是，ip netns exec命令允许我们以root权限进入指定的虚拟网络空间进行自定义配置。下面的命令使用ip netns exec为net0网络空间中的网络设备eth0设置IP地址：

```
# ip netns exec net0 ip addr add 172.17.42.99/16 dev eth0
```

在12.1.5节中，我们会讲解如何在不依赖Docker引擎的情况下利用这种技术为容器纯手工打造一个虚拟的网络环境。

12.1.3　网络设备虚拟化

如果一台计算机想通过IP协议与其他主机通信，它势必需要一个网络接口，也就是我们通常所说的网卡。另外，还需要一个将各个主机连接在一起的网络设备。这台设备通过内部存储的路由表控制数据包的发送方向，它可能是网桥、路由器、交换机或者其他类似的网络连接设备。这与网卡需要连接到一个路由器上才能与其他主机进行连接类似。

如图12-1所示，主机A与主机B通过网卡与路由器相连，路由器与广域网相连，这样主机A与主机B不但可以通过路由器互相通信，还能通过它访问广域网的内容。

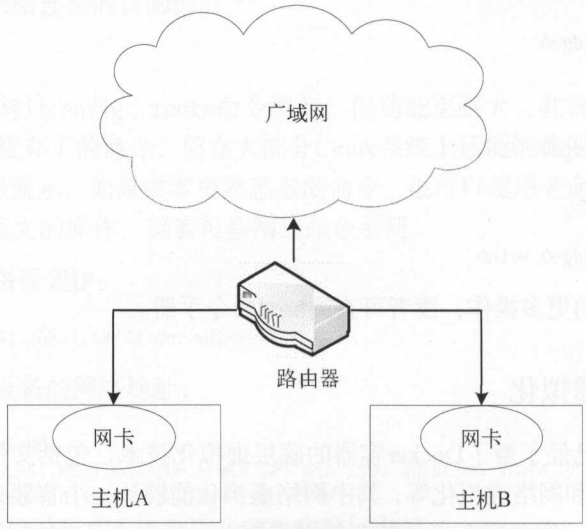


图12-1　网络基本连接

此外，网络接口和网络设备都不一定是物理设备，也可以是虚拟设备，比如每个Linux系统的lo回环接口就是一个虚拟的网络接口，它主要用于快速高效地发送和接收回环包。Docker就是利用特殊的虚拟网络接口来连接容器和主机的。

当我们在主机上启动Docker服务时，Docker引擎会在操作系统的内核创建一个名为docker0的虚拟以太网桥，并且为这个网桥设备随机分配一个主机没有使用的私有IP地址和子网掩码。这

个网桥将连接在它上面的所有容器组织成一个虚拟子网，并且提供数据包转发功能。可以认为，这个虚拟网桥是连接容器与容器、容器与主机之间的路由中枢。

下面的ip addr命令展示了笔者机器上的一个无线网卡3: wlan0和Docker创建的虚拟网桥4:docker0：

```
# ip addr
...
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 48:5d:60:ed:5a:0d brd ff:ff:ff:ff:ff:ff
    inet 172.16.9.18/24 brd 172.16.9.255 scope global wlan0
       valid_lft forever preferred_lft forever
    inet6 fe80::4a5d:60ff:feed:5a0d/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

其中wlan0的IP地址是172.16.9.18/24，docker0的IP地址是172.17.42.1/16。新启动的容器默认都将连接到docker0网桥之上。

此时虚拟的桥接路由设备已经有了，我们还需要虚拟的网卡设备提供给容器使用。在Linux中，我们可以创建虚拟的以太设备（veth），英文全名为Virtual Ethernet，它类似于我们的物理网卡设备。这种新的设备类型由Linux容器技术所引进。此外，veth设备必须成对出现。下面的命令用于创建一对veth设备：

```
# ip link add A type veth peer name B
```

上面创建的一对veth设备的名字分别为A和B。因为必须成对出现，所以删除其中任意一个，另外一个也会被删除。veth的两端设备除了名字不一样，两个设备是完全对称的，从一个设备发出的数据会从另外一个设备收到，在一个设备上的操作会完全映射到另一个设备之上。默认情况下，这两个设备都处于主机的网络空间中，但一般会将一个设备置于容器的网络空间，另一个根据需要可以放到其他容器的网络空间或者主机的网络空间。

Docker容器的网络互联也正是依赖Linux底层的veth机制。每当启动一个容器时，Docker引擎会在主机上创建一个虚拟的以太网卡接口，并且会将这个veth接口一端放在主机网络空间并连接到docker0网桥，另一端放入Docker容器的网络空间中。容器退出时，虚拟网卡也会被销毁，下次启动容器时将创建一个新的虚拟网络接口并与docker0重新连接。在运行两个容器的主机上面，下面的ip addr命令展示了Docker创建的两个虚拟网卡设备：

```
# ip addr
...(略)
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
...(略)
```

```
6: vethb926edc: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master docker0 state UP group default
qlen 1000
    link/ether e6:ab:24:d0:a1:bd brd ff:ff:ff:ff:ff:ff
    inet6 fe80::e4ab:24ff:fed0:a1bd/64 scope link
       valid_lft forever preferred_lft forever
7: vethcff2cc8: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master docker0 state UP group default
qlen 1000
    link/ether 2e:89:46:af:6e:b5 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::2c89:46ff:feaf:6eb5/64 scope link
       valid_lft forever preferred_lft forever
```

主机端的虚拟网卡设备一般是以veth*的形式命名的，如上面的vethb926edc和vethcff2cc8。

下面我们使用brctl show命令来验证vethb926edc和vethcff2cc8这两个网络接口是否连接在docker0网桥之上：

```
# brctl show
bridge name  bridge id              STP enabled     interfaces
docker0      8000.56847afe9799      no              vethb926edc
                                                    vethcff2cc8
```

其中interfaces字段列出了连接在容器上的所有网络设备名称。

veth置于容器内部的一端设备会被Docker命名为eth0，供容器内的进程使用，用来与外部网络进行通信。容器内的eth0和主机上的veth*相当于一根管道的两端，数据包从一端经由管道传递到另一端，从而打通了容器内部与外部的通信。图12-2展示了主机网络接口端vethb926edc与容器网络接口端eth0通过管道相连。

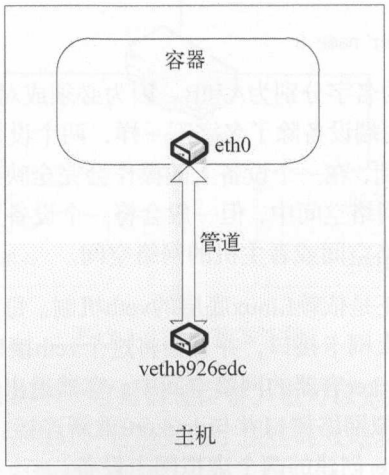


图12-2　网卡管道示意图

我们进入到其中一个容器，使用ip addr命令查看eth0接口信息：

```
root@338de75b79e5:/# ip addr
...(略)
2: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 02:42:ac:11:00:06 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:6/64 scope link
       valid_lft forever preferred_lft forever
```

可以看到，IP地址是172.17.0.3/16，证明其处于docker0网桥的子网之下。

以上容器的网络结构如图12-3所示。

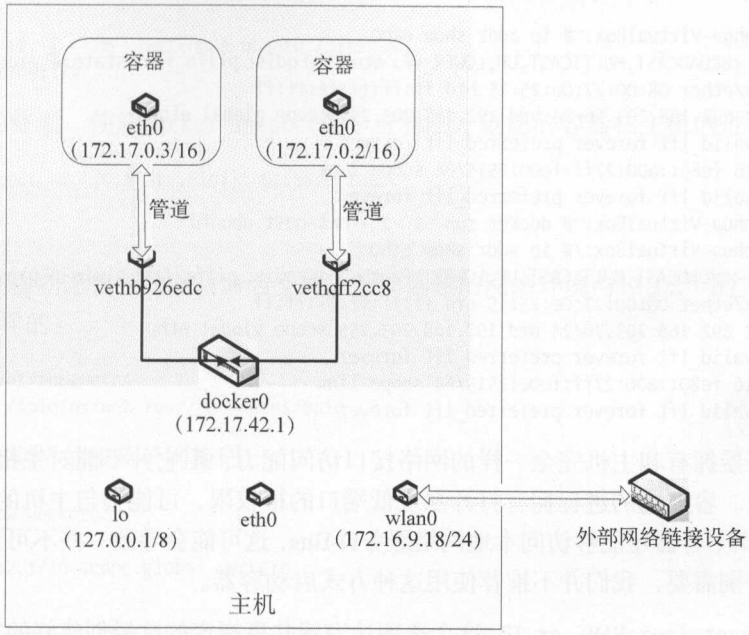


图12-3 主机内容器网络结构图

12.1.4 容器运行的4种网络模式

除了上一节所讲的一般情况外，即让容器运行在通过创建虚拟网卡网桥配置而成的网络环境中，我们还可以让容器运行在主机或者其他容器的网络环境中，甚至可以剔除掉容器的网络能力。在使用docker run命令运行容器时，我们通过--net参数来指定容器运行的网络模式，一共有4种模式可选，具体如下所示。

- --net=bridge。这是容器运行的默认网络模式。通过网桥来设置容器网络时，默认会连接到docker0网桥之上。此外，也可以通过全局配置文件里DOCKER_OPTS选项的-b参数来指定默认网桥。

假如我们希望容器连接到一个名为docker1的网桥上，只要在/etc/default/docker文件的DOCKER_OPTS选项里添加-b=docker1，然后重启docker服务即可。如果docker1是一个非法网桥，docker服务重启将失败。配置如下：

```
DOCKER_OPTS="-b=docker1"
```

- --net=host。此网络模式的容器将不会拥有自己独立的网络空间，他处的网络环境将不再是虚拟化的了。相反，它会共享主机的网络环境，包括IP地址、网卡设备、主机名、网络配置等一切与网络有关的资源。但容器在进程、文件系统等虚拟化方面并不会受到影响。下面的代码验证了容器确实与主机处于同一网络环境中：

```
root@xinhua-VirtualBox:~# ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:0e:25:15 brd ff:ff:ff:ff:ff:ff
    inet 192.168.203.76/24 brd 192.168.203.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe0e:2515/64 scope link
       valid_lft forever preferred_lft forever
root@xinhua-VirtualBox:~# docker run -t -i --net=host ubuntu
root@xinhua-VirtualBox:/# ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:0e:25:15 brd ff:ff:ff:ff:ff:ff
    inet 192.168.203.76/24 brd 192.168.203.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe0e:2515/64 scope link
       valid_lft forever preferred_lft forever
```

虽然容器拥有和主机完全一样的网络接口访问能力，但它并不能对主机的网络配置栈进行修改。容器内的进程拥有打开系统低端口的根权限，可能会与主机的某些进程产生冲突。另外，容器还能够访问本地网络服务D-Bus，这可能会导致一些不可预料的问题发生。除非特别需要，我们并不推荐使用这种方式启动容器。

- --net=container:NAME_or_ID。这个选项让容器共享指定的已经创建好的容器的网络环境。两个容器将共享IP地址及端口号等网络资源，并且这两个容器的进程可以通过回环网络进行访问。

  先在一个终端启动一个容器：

```
# docker run -t -i --name=theOne Ubuntu
root@167b6fc7f62c:/ #
```

  在另一个终端使用--net=container:167b6fc7f62c参数启动另一个容器：

```
# docker run -t -i --name=theTwo --net=container:167b6fc7f62c ubuntu
root@167b6fc7f62c:/ #
```

  这样theTwo容器就共用了theOne容器的网络环境了，包括它的主机名。

- --net=none。以这个选项运行的容器将不会被配置网络环境，也就是说，它将不具备任何网络访问能力。对于已经使用该配置运行的容器，我们仍然可以手动为它配置一个网络环境。大部分情况下，我们并没必要这么做，因为Docker引擎已经为我们做好了。

12.1.5 手动配置容器的网络环境

下面我们将手动为运行在--net=none参数下的容器配置一个虚拟的网络环境，以帮助读者详细了解Docker的网络原理，也算是本节的一个小结。

(1) 在一个终端中用--net=none选项运行一个无网络环境的容器：

```
# docker run -i -t --rm --net=none ubuntu /bin/bash
root@63f36fc01b5f:/#
```

(2) 启动新的终端，使用docker inspect命令查看刚才创建的容器在主机的进程ID：

```
# docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
# pid=2778
```

(3) 为容器进程创建独立的网络命名空间。下面创建的网络环境与12.1.2节的方式不一样，但达到的效果是一样的：

```
# mkdir -p /var/run/netns
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

(4) 查看默认网桥的IP及子网掩码：

```
# ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...(略)
```

(5) 创建虚拟网络接口设备A，并为A创建一个映射端设备B：

```
# ip link add A type veth peer name B
```

(6) 将设备A接入网桥docker0：

```
# brctl addif docker0 A
```

(7) 启动网络设备A：

```
# ip link set A up
```

(8) 将B设备放到刚创建的独立的网络空间中去：

```
# ip link set B netns $pid
```

(9) 在虚拟网络空间中将设备B重命名为更通用的名称eth0：

```
# ip netns exec $pid ip link set dev B name eth0
```

(10) 为eth0设置MAC地址：

```
# ip netns exec $pid ip link set eth0 address 12:34:56:78:9a:bc
```

(11) 启动eth0：

```
# ip netns exec $pid ip link set eth0 up
```

(12) 为eth0配置网络IP地址及子网掩码：

```
# ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
```

(13) 为主机设置路由地址：

```
# ip netns exec $pid ip route add default via 172.17.42.1
```

至此，容器的网络环境已经配置完毕。可以看到，它与Docker默认为我们配置的网络环境一样。

出于安全性考虑，容器的进程是不允许更改它自己的网络配置的，所以ip netns exec所执行的网络命令放到Docker容器中执行可能并不会生效。我们可以使用--privileged=true赋予容器更改网络配置的权限，但这样会增加容器网络在内部被破坏的风险。相比而已，我们更加推荐使用ip nets exec这种更加安全的方式去配置网络环境。

另外需要注意的是，Docker引擎并不会保存我们对容器所做的网络环境配置。当容器退出时，我们创建的虚拟网络空间也会被销毁，包括放置在网络空间的eth0和主机端的设备A。

12.2　配置及原理

在这一节中，我们会讲到Docker网络的基本配置。结合上一节的网络原理，我们会深入配置后的原理。

12.2.1　基本配置

Linux主机名保存在/etc/hostname文件中，域名解析服务器IP保存在/etc/resolv.conf文件中，host配置保存在/etc/hosts文件中。Docker容器也是一样的，我们看一下使用默认参数启动一个容器的这3个文件：

```
# docker run -t -i ubuntu
root@c85588436954:/# cat /etc/hostname
c85588436954
root@c85588436954:/# cat /etc/hosts
172.17.0.2 c85588436954
127.0.0.1  localhost
::1 localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
root@c85588436954:/# cat /etc/resolv.conf
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#    DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN

nameserver 8.8.8.8
nameserver 8.8.4.4
```

接下来，我们将介绍如何修改这些默认值以及这些修改背后的原理。

1. 主机名配置

默认情况下，会以容器ID的前12个字符作为主机名。当然，我们也可以使用-h HOSTNAME或--hostname=HOSTNAME选项来修改主机名：

```
# docker run -h myhostname -t -i ubuntu
root@myhostname:/# cat /etc/hostname
myhostname
root@myhostname:/# cat /etc/hosts
172.17.0.2 myhostname
127.0.0.1  localhost
...(略)
```

-h选项将直接修改/etc/hostname文件，同时也会修改/etc/hosts主机的IP对应的主机名。容器的主机名会显示在容器内/bin/bash终端的计算机名区域（终端@符号后的区域），而它对主机和其他容器是不可见的。在主机上使用docker ps命令，也无法看到容器的主机名。

2. DNS配置

使用参数--dns=IP_ADDRRESS可以为新启动的容器配置一个或者多个域名服务器，这个参数会修改/etc/resolv.conf文件。容器中的进程需要通过主机名或域名来访问时，如果该域名在/etc/hosts文件中不能被解析，就会从文件/etc/resolv.conf中读取DNS服务器的IP地址，然后在端口53上向DNS服务器请求解析域名到IP地址。

--dns-search=DOMAIN用来配置DNS查询的后置域。举个例子，如果你设置DNS搜索域为baidu.com，那么当你访问主机名为music的主机时，容器将不仅仅去DNS服务器查询music的IP地址，还会同时查询music.baidu.com域名的IP地址。如果不想使用这个设置，将DOMAIN置为空即可。这个配置同样会修改/etc/resolv.conf文件。

Docker从版本1.2.0开始，支持在容器中直接修改/etc/hostname、/etc/hosts和/etc/resolve.conf这3个文件。不过对文件的修改也仅在本次容器运行期间有效，容器退出后修改也随之丢失。通过docker commit命令将容器运行环境保存成镜像时，这3个文件的修改也会丢失而不会写入到容器镜像中去。

Docker容器的主机名和DNS等网络配置并不是在镜像中修改的，而是在启动的那一刻通过覆写下面这3个文件来达到：

```
/etc/hostname
/etc/hosts
/etc/resolv.conf
```

在容器中使用mount命令可以发现，hostname、hosts和resolv.conf这3个文件其实是挂载在主机上的文件，每一个容器对应着一份副本，通过uuid进行隔离：

```
root@myhostname:/# mount
...(略)
/dev/disk/by-uuid/ff907709-9cd2-4189-ae46-67e2df64753e on /etc/resolv.conf type ext4
(rw,relatime,errors=remount-ro,data=ordered)
/dev/disk/by-uuid/ff907709-9cd2-4189-ae46-67e2df64753e on /etc/hostname type ext4
(rw,relatime,errors=remount-ro,data=ordered)
/dev/disk/by-uuid/ff907709-9cd2-4189-ae46-67e2df64753e on /etc/hosts type ext4
(rw,relatime,errors=remount-ro,data=ordered)
...(略)
```

我们不应该直接修改这些文件来达到目标，Docker会为我们自动配置好这些文件，以满足当前网络的需求。如果需要修改，需要在首次运行容器时通过命令进行配置。

12.2.2　容器互联配置及原理

上一章中，我们已经讲过容器间的互联。容器与容器之间能否进行通信，主要由下面两个因素所决定。

- **需要通信的容器的网络接口是否处于同一网络拓扑结构中。**默认情况下，Docker会将容器接入到名为docker0的网桥上。网桥会处理容器之间的数据包交换。
- **iptables配置是否允许两个容器之间创建连接。**

当我们启动容器时，Docker引擎会在主机iptables中创建一条转发链。如果我们配置了--icc=true，这条转发链就使用ACCEPT策略进行配置。如果--icc设置成了false，就使用DROP策略进行配置。将--icc参数设置成true还是false取决于我们是否要依赖iptables来保护容器免受端口刺探或被其他无关容器访问的危险。

分别在--icc设成true和false的情况下，使用下面的命令创建一个容器：

```
# docker run -t -i -p 80 ubuntu
```

- --icc=false时，过滤表的规则如下：

```
# iptables -L -n
...(略)
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.2           tcp dpt:80
```

```
DROP       all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
...( 略)
```

上面的规则表明，目标地址为容器的IP地址172.17.0.2，tcp端口号为80的数据都将被丢弃。

- --icc=true时，过滤表的规则如下：

```
# iptables -L -n
...(略)
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.2   tcp dpt:80
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0    ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
...( 略)
```

上面的规则表明，目标地址为容器的IP地址172.17.0.2，tcp端口号为80的数据都将被转发。

--icc参数通过全局配置文件/etc/default/docker中的DOCKER_OPTS选项进行配置，如：

```
DOCKER_OPTS="--icc=false"
```

或者

```
DOCKER_OPTS="--icc=true"
```

通过设置全局配置中DOCKER_OPTS选项中的--iptables为false，启动容器将不会修改主机的iptables。一般情况下，我们不需要这么做，否则容器网络将不能正常工作。

为了保证最大的网络安全，一般选择将--icc设置成false，那么这种情况下我们如何能让容器之间进行通信呢？这就要依赖第4章所讲的--link参数。在将--icc置为false，iptables置为true，而又指定了--link参数的情况下，Docker会在iptables里插入一条ACCEPT规则，以允许容器能访问另外一个容器所打开的端口，这个端口可由镜像在Dockerfile文件通过EXPOSE命令指定，也可以是使用-p参数打开的端口。下面我们将演示这种情况。

(1) 配置--icc=false并重启docker服务：

```
# echo 'DOCKER_OPTS="--icc=false"' > /etc/default/docker
# service docker restart
```

(2) 在一个终端启动一个开放80端口的容器：

```
# docker run -t -i -p 80 --name=gloomy_fermi ubuntu
```

(3) 在另一个终端启动另一容器：

```
sudo docker run -t -i --link=gloomy_fermi:sv Ubuntu
```

其中gloomy_fermi是第一个容器的名称。

(4) 在第3个终端查看iptables规则：

```
# iptables -L -n
...(略)
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.3           172.17.0.4          tcp spt:80
ACCEPT     tcp  --  172.17.0.4           172.17.0.3          tcp dpt:80
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...(略)
```

上面的规则表明，允许172.17.0.4容器1与172.17.0.3容器2在容器2的tcp端口80上进行通信。

--link=gloomy_fermi:sv选项会在/etc/hosts文件中添加一项名为sv的域名解析。这个sv对应的IP即gloomy_fermi容器对应的IP地址。这样容器内的进程在不知道gloomy_fermi容器的IP地址的情况下，也可以使用sv别名与之通信。当容器重启的时候，可能会获得与上一次不一样的IP地址，Docker会自动修改接收者容器/etc/hosts文件中的sv项目，以保证通信正常。查看上例接收者容器中/etc/hosts的内容：

```
# cat /etc/hosts
172.17.0.4 7c1944a258c3
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3 sv
```

其中172.17.0.3 sv项是通过--link=gloomy_fermi:sv选项添加进来的。

12.2.3　容器内访配置及原理

默认情况下，容器能与外部创建连接，但外部网络不能主动连接容器。如果希望容器能够接收来自外部的连接，可以在使用docker run命令运行容器时指定一个特殊参数，即-p或-P。-p和-P都是用来向外部开放内访的端口的。

- 使用-P或者--publish-all=true|false。有了这个参数，创建容器镜像的Dockerfile文件中EXPOSE命令所指定的端口都将变成可访问的。这个命令会将EXPOSE的端口列表随机映射到主机49153~65535的端口范围。如果想知道具体是哪个端口，还需要额外的命令操作。
- 使用-p SPEC或--publish=SPEC（这个方法更方便）。这个参数不仅允许我们添加可内访的端口，还可以指定该端口的类型（TCP、UDP）及其在主机上映射的具体端口。

无论使用哪种方法，最终都需要通过修改主机上iptables的nat表来完成。

假如使用docker run -t -i -p 80 ubuntu启动一个容器，则nat表的形式如下：

```
# iptables -t nat -L -n
...(略)
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:49155 to:172.17.0.5:80
...(略)
```

上面的动态地址转换（DNAT）规则将匹配任何流经本机且目的端口为49155的tcp&ip包，并将该包的目标地址及端口转换成容器网络的IP地址172.17.0.5及容器开放的端口80。经过这样的转换，外部网络的包就能顺利抵达容器内部的网络。

下面我们使用docker run -t -i -p 127.0.0.1:80:8080/udp ubuntu命令来启动一个容器，这是一个稍微复杂一点的端口映射例子，指明的是将容器的udp端口80映射到主机的localhost（127.0.0.1）网络接口上的udp端口8080上。这种情况下，nat表的形式如下：

```
# iptables -t nat -L -n
...(略)
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       udp  --  0.0.0.0/0            127.0.0.1           udp dpt:80 to:172.17.0.6:8080
...(略)
```

我们可以看到DNAT规则的协议类型（prot）匹配变成了udp，目标地址（destination）匹配变成了127.0.0.1，转换的目标端口变成了我们指定的8080端口。

另外，我们可以修改Docker全局配置文件/etc/default/docker的DOCKER_OPTS选项，通过配置参数--ip=IP_ADDRESS，达到指定一个或多个目标IP地址的作用。记住，修改全局配置时，需要重启Docker服务方可生效。

12.2.4 容器外访配置及原理

是否允许容器访问外部网络，主要受主机的ip_forward系统参数影响。默认情况下，ip_forward设置成1，如果它被设置成0，则可以通过以下方法打开它：

```
# cat /proc/sys/net/ipv4/ip_forward
0
# echo 1 > /proc/sys/net/ipv4/ip_forward
# cat /proc/sys/net/ipv4/ip_forward
1
```

如果我们将Docker服务的参数--ip-forward设成true，那么Docker服务每次启动时，都会将/proc/sys/net/ipv4/ip_forward设成1。如果是false，则启动服务时不会对主机ip_forward做任何修改。大多数情况下，我们可以打开这个开关，以确保与外部网络及其他容器间正常通信。

每一个从容器发往外部的源IP地址最终都会被修改为主机的IP地址。这得益于iptables的MASQUERADE规则。当Docker服务启动时，就会在iptables中添加一条这样的规则。

下面的MASQUERADE规则表明所有的源地址为172.17.0.0/16的数据包在路由前数据包中的源IP地址字段都会被替换成主机的IP地址：

```
# iptables -t nat -L -n
...(略)
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
... (略)
```

12.2.5　创建点对点连接

默认情况下，Docker会将所有的容器接入到虚拟网桥docker0所组织的虚拟子网中。我们已经学过如何将容器连接到创建的网桥之上，但是有时候我们仅仅需要两个容器之间能够直接通信，而不想让它们混入其他复杂的网络中。

解决方案很简单，当我们创建一对veth设备时，将veth的两端网络接口设备分别放入两个容器中，并且就像配置普通网络接口那样配置它们。这样，两个容器就能直接进行通信了。图12-4展示了这种解决方案。下面我们将演示具体的创建步骤。

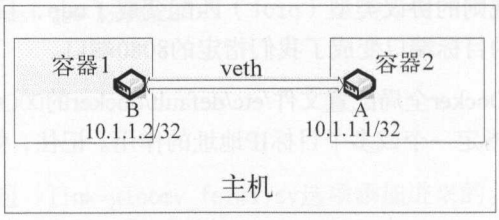


图12-4　容器点对点连接

(1) 在终端1创建一个无网络的容器767318ce80c3，下面将以容器1指代：

```
# docker run --rm -i -t --net=none ubuntu
root@767318ce80c3:/#
```

(2) 在终端2创建一个无网络的容器bd1f6fe8f7c7，下面将以容器2指代：

```
# docker run --rm -i -t --net=none ubuntu
root@bd1f6fe8f7c7:/#
```

(3) 在终端3查看刚才创建的两个容器的进程ID：

```
# docker inspect -f '{{.State.Pid}}' 767318ce80c3
4207
# docker inspect -f '{{.State.Pid}}' bd1f6fe8f7c7
4241
```

(4) 在终端3分别为两个容器在它们独立的进程空间里创建虚拟网络空间：

```
# mkdir -p /var/run/netns
# ln -s /proc/4207/ns/net /var/run/netns/4207
# ln -s /proc/4241/ns/net /var/run/netns/4241
```

(5) 创建一对veth，两端命名分别为A、B：

```
# ip link add A type veth peer name B
```

(6) 将A放入容器1的网络空间：

```
# ip link set A netns 4207
```

(7) 在容器1的网络空间中设置设备A的IP地址为10.1.1.1/32：

```
# ip netns exec 4207 ip addr add 10.1.1.1/32 dev A
```

(8) 在容器1的网络空间中启动设备A：

```
# ip netns exec 4207 ip link set A up
```

(9) 在容器1的网络空间中设置设备A的路由地址为10.1.1.2/32：

```
# ip netns exec 4207 ip route add 10.1.1.2/32 dev A
```

(10) 将B放入容器1的网络空间：

```
# ip link set B netns 4241
```

(11) 在容器2的网络空间中设置设备B的IP地址为10.1.1.2/32：

```
# ip netns exec 4241 ip addr add 10.1.1.2/32 dev B
```

(12) 在容器2的网络空间中启动设备B：

```
# ip netns exec 4241 ip link set B up
```

(13) 在容器2的网络空间中设置设备B的路由地址为10.1.1.1/32：

```
# ip netns exec 4241 ip route add 10.1.1.1/32 dev B
```

现在这两个容器就能直接通信了。点对点通信不需要依赖子网，所以也不需要子网掩码。我们需要做的仅仅是通过ip route命令为veth在本端的设备指定路由地址为对端的IP地址。点对点连接可以与其他网络连接方式安全共存，这就意味我们没有必要以--net=none方式启动一个容器了。

我们同样可以将这种点对点的连接方式应用到容器与主机之间，这样允许主机使用一个独立IP地址与那个容器进行通信。这种不依赖网桥的通信方式可以用来关闭容器与其他容器之间的通信，但是除非有特殊的网络需求，我们更加推荐使用12.2.2节介绍的--icc=false参数来达到关闭容器间通信通道的目的。

12.3　网桥

前面已经讲到网桥在容器网络中所发挥的巨大作用，本节中我们将学习网桥的配置以及如何为容器创建我们自己的网桥。

12.3.1　配置网桥

在默认网络模式（--net=bridge）下启动容器时，容器会检查Docker服务的全局配置中是否指定了-b参数，如果指定了，容器就连接到这个参数所指定的网桥，没有指定就连接到默认的网桥docker0。

docker0网桥的IP地址、子网掩码、MTU值及连接设备IP地址分配范围是由Docker服务配置的，我们可以通过全局配置/etc/default/docker文件中的DOCKER_OPTS选项来修改这些值。以下是具体的修改参数说明。

- --bip=CIDR。为默认网桥指定一个子网掩码和IP地址。需要使用标准CIDR格式，如192.168.1.5/24。
- --fixed-cidr=CIDR。设定docker0网桥下的子网IP分配范围。需要使用标准CIDR格式，如172.167.1.0/28。
- --mtu=BYTES。指定网桥的最大传输单元长度。如果不指定，则使用默认值1500字节。

修改/etc/default/docker文件之后，需要重启Docker服务配置方可生效。

12.3.2　构建自己的网桥

通常情况下，Docker创建的默认网桥已经能够满足需求。但我们仍可以在不依赖Docker的基础上，完完全全手工为容器创建一个网桥，并通过参数-b或者--bridge告诉Docker使用我们自定义的网桥。下面我们将一步一步讲解如何构建网桥。

(1) 停止已经启动的服务：

```
# sudo service docker stop
```

(2) 停止默认网桥设备docker0：

```
# sudo ip link set dev docker0 down
```

(3) 删除默认网桥：

```
# sudo brctl delbr docker0
```

(4) 创建虚拟网桥bridge0：

```
# sudo brctl addbr bridge0
```

(5) 为新建的网桥设定IP地址和子网掩码：

```
# sudo ip addr add 192.168.5.1/24 dev bridge0
```

(6) 启动新建网桥：

```
# sudo ip link set dev bridge0 up
```

(7) 为了确认网桥是否如期启动，通过下面的命令进行检视：

```
# ip addr show bridge0
3: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
      valid_lft forever preferred_lft forever
```

(8) 修改全局配置，设定bridge0为默认网桥：

```
# echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
```

(9) 重启Docker服务：

```
$ sudo service docker restart
```

至此，我们自己的网桥就创建好了，之后运行的容器都将会连接到这个网桥上。

第13章 安 全

我们知道Docker是基于操作系统级的虚拟技术，而虚拟机是基于硬件层面的虚拟技术。相对而言，Docker的安全性一直受到怀疑，即便Docker看起来像沙箱一样安全。不过Docker公司后面也承认，容器的安全是今后需要重点加强的部分。事实上，Docker也正在行动，它和Red Hat组件安全小组一起在加强安全性能。Docker的安全机制主要依赖Linux已有的安全机制，恰如其分地使用已有的技术往往比推倒一切重来来得简单和巧妙，这和Linux的工具链颇为相似。本章的主要内容有：

- 命名空间。
- cgroups。
- Linux能力机制。

13.1 命名空间

Linux的命名空间对虚拟化提供了轻量级的支持，通过它我们可以完全隔离不同的进程。以往，在Linux及其他类UNIX系统中，很多资源都是全局的，包括进程号（Pid）、用户信息、系统信息、网络接口和文件系统等。用户可以看到其他用户的进程、使用的一些资源等。多数情况下，这都没问题。但在有些时候，这不能满足我们的需求。如果服务器供应商向客户提供Linux计算机的全部访问权限的话，那么在传统的做法中，可能要为每个用户提供一台计算机，这样做代价太高，而且计算机也不能完全发挥作用。使用虚拟机是一种解决方案，但是资源的利用率还是太低。每个虚拟机都需要一个独立的系统，安装配套的应用层应用，这会占用大量的磁盘空间。有多个内核都在同时运行，由于虚拟机内核对程序指令封装了一层，执行效率也会大打折扣。

命名空间提供了一种不同的解决方案，只占用很少的资源，只需要运行计算机本身的操作系统。所有的进程都在同一个系统上运行，需要隔离的各种资源则通过命名空间达到隔离的目的。这样就可以把一些进程放到一个容器中，另一些进程放到另一个容器中，两个容器之间互相隔离。当然，也可以根据需要允许容器间有一定的共享。例如，容器使用独立的PID集合，但是和其他容器共享文件系统。本质上，命名空间提供了对资源的不同视图，在不同的命名空间下我们会看到不同的资源集合。之前的每一项全局资源都被封装到容器的数据结构中。只有资源和包含资源

的命名空间构成的组合是全局唯一的。也许在容器内部资源是唯一的，但是从容器外部看就保证不了了。

图13-1显示了命名空间如何隔离进程。所有的命名空间都在同一个内核上运行，命名空间1中有进程1-1和进程1-2，命名空间2中有进程2-1和进程2-2，命名空间n中有进程n-1和进程n-2。在进程1-1中可以看到进程1-2，但是看不到进程2-1，也看不到进程n-1。每个命名空间中的进程都认为它们独占整个系统。

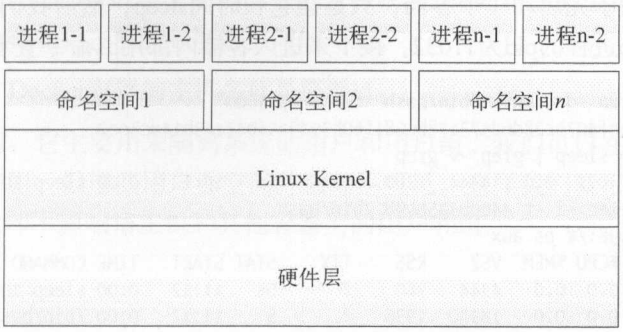


图13-1 命名空间将进程隔离

图13-2演示了系统上有3个命名空间的情况。一个命名空间是父命名空间，它衍生了两个子命名空间。假定容器用于虚拟主机配置，其中每个容器看起来必须像是单独的一台Linux计算机。因此，其中每一个都有自身的init进程，PID为1，其他进程的PID以递增次序分配。两个子命名空间都有PID为1的init进程，以及PID分别为2和3的两个进程。由于相同的PID在系统中会出现多次，所以PID号不是全局唯一的。虽然子容器不了解系统中的其他容器，但父容器知道子命名空间的存在，而且可以看到其中执行的所有进程。图中子容器的进程映射到父容器中，PID为4到9。尽管系统上有9个进程，但却需要15个PID来表示，因为一个进程可以关联到多个PID。至于哪个PID是“正确”的，则依赖于具体的上下文。

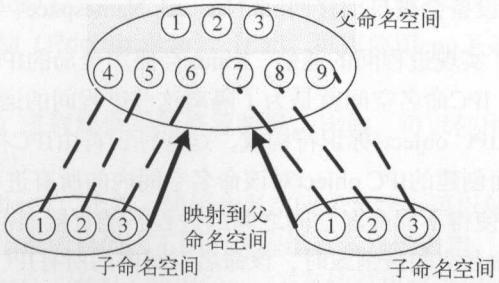


图13-2 命名空间的层次关联

不同的资源构成了不同的命名空间，这里简要介绍一下进程命名空间、网络命名空间、IPC命名空间、挂载命名空间、UTS命名空间和用户命名空间这6种。

- **进程命名空间（PID Namespace）**。该命名空间主要用来管理进程id以及其他id（tgid、pgid和sid）。这里我们以进程id为例来说明进程命名空间的作用。在创建进程时，Linux会为它分配一个号码以在其命名空间中唯一标识它，该号码称作进程id号，常用pid表示。同一进程在不同的进程命名空间下会有不同的进程id，每个进程命名空间可以按自己的方法管理进程id。所有的进程命名空间组成一个树形结构，子空间中的进程对于各级父辈空间是可见的，进程在各可见空间中都有一个进程id与之对应。

  下面的代码用于先启动一个容器，只是让它长时间sleep，然后我们在宿主机上通过ps命令查看到sleep进程的pid为11032，接下来进入容器内部用ps命令查看sleep进程的pid为1：

```
$ sudo docker run -d ubuntu /bin/bash -c "sleep 1000"
440db3a5bb98a6acf407bc28d6d573178bc081f60b3949a9fb43c43b14dc3ce2
$ ps aux | grep sleep | grep -v grep
root     11032  0.1  0.0   4344   360 ?        Ss   19:12   0:00 sleep 1000
$ sudo docker exec -i -t 440db3a5bb98 /bin/bash
root@440db3a5bb98:/# ps aux
USER       PID %CPU %MEM  VSZ    RSS    TTY    STAT START   TIME COMMAND
root       1   0.0  0.0   4344   360    ?      Ss   11:12   0:00 sleep 1000
root       7   0.0  0.0   18140  1936   ?      S    11:12   0:00 /bin/bash
root       22  0.0  0.0   15568  1124   ?      R+   11:13   0:00 ps aux
root@440db3a5bb98:/#
```

- **网络命名空间（Network Namespace）**。该命名空间为进程提供了一个完全独立的网络协议栈的视图，包括网络设备接口、IPv4和IPv6协议栈、IP路由表、防火墙规则和sockets等。网络命名空间提供了一份独立的网络环境，就跟一个独立的系统一样。物理设备只能存在于网络命名空间中。通过给每个容器建立一个独立的网络命名空间，可以为容器提供一个虚拟的独立的网络环境，就好像自己有一个私有的网络接口一样。

  虚拟网络设备（virtual network device）还提供了一种类似管道的抽象，可以在不同的命名空间之间建立隧道。利用虚拟化网络设备，可以建立到其他命名空间中的物理设备的桥接。利用这种桥接，我们可以实现容器间的网络通信。当一个网络命名空间被销毁时，该命名空间的物理设备会被自动移回init Network Namespace，即系统最开始的命名空间。

- **IPC命名空间**。为了实现进程间的通信，Linux会使用全局的IPC对象，而所有进程都可以见到这些IPC对象。IPC命名空间就是为了隔离这些进程间的通信资源的。一个IPC命名空间由一组System V IPC objects标识符构成，这些标识符由IPC相关的系统调用创建。在一个IPC命名空间里面创建的IPC object对该命名空间内的所有进程可见，但是对其他命名空间不可见，这样就使得不同命名空间之间的进程不能直接通信，就像是在不同的系统里一样。当一个IPC命名空间被销毁时，该命名空间内的所有IPC object会被内核自动销毁。

  PID命名空间和IPC命名空间可以组合起来一起使用，这样新创建的命名空间既是一个独立的PID空间，又是一个独立的IPC空间。不同命名空间的进程彼此不可见，也不能互相通信，这样就实现了进程组间的隔离。

- **挂载命名空间**。挂载命名空间为进程提供了一个文件层次视图，每个进程都存在于一个挂载命名空间里。默认情况下，子进程和父进程将共享同一个挂载命名空间，其后子进程调用mount或umount将会影响到所有该命名空间内的进程。如果子进程在一个独立的挂载命名空间里，就可以调用mount或umount命令建立一份新的文件层次视图。这样的话，不同的容器就拥有独立的文件系统了。
- **UTS（UNIX Time-sharing System）命名空间**。该命名空间主要用来管理主机名和域名。每个UTS命名空间都可以定义不同的主机名和域名。通过配置独立的UTS命名空间，就可以虚拟出一个有独立主机名和网络空间的环境。

  默认情况下，Docker容器的主机名就是容器id。
- **用户命名空间**。它主要用来隔离系统的用户和用户组。我们可以在用户命名空间中建立自己的用户和组，但这些用户在空间外面却不可见。这样我们就可以在容器中自由地添加用户和组，却不影响宿主机和其他容器上的用户和组。

13.2 cgroups

cgroups是control groups的缩写，是Linux内核提供的一种可以记录、限制、隔离进程组（process group）所使用的物理资源（如CPU、内存、I/O等）的机制。它最初由Google工程师提出，后来被整合进Linux内核。cgroups也是容器为实现虚拟化所使用的资源管理手段。可以说，没有cgroups，就没有容器。

cgroups最初的目标是为资源管理提供一个统一的框架，既整合现有的cpuset等子系统，也为未来开发新的子系统提供接口。现在的cgroups适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化（OS Level Virtualization）。cgroups提供了以下功能。

- **限制进程组可以使用的资源数量（Resource limiting）**。比如，memory子系统可以为进程组设定一个memory使用上限，一旦进程组使用的内存达到限额再申请内存，就会出现OOM（Out Of Memory）。
- **进程组的优先级控制（Prioritization）**。比如，可以使用cpu子系统为某个进程组分配特定CPU占有率。
- **记录（Accounting）进程组使用的资源数量**。比如，可以使用cpuacct子系统记录某个进程组使用的CPU时间。
- **进程组隔离（Isolation）**。比如，使用命名空间子系统，可以使不同的进程组使用不同的命名空间，以达到隔离的目的，不同的进程组有各自的进程、网络、文件系统挂载空间。
- **进程组控制（Control）**。比如，使用freezer子系统可以将进程组挂起和恢复。

控制组是Linux容器机制的另外一个关键组件，负责实现资源的审计和限制。它提供了很多有用的特性，确保各个容器可以公平地分享主机的内存、CPU、磁盘I/O等资源。当然，更重要的是，控制组确保了当容器内的资源使用产生压力时，不会连累主机系统。

尽管控制组不负责隔离容器之间相互访问、处理数据和进程，它在防止拒绝服务（DDOS）攻击方面是必不可少的。尤其是在多用户的平台（比如公有或私有的 PaaS）上，控制组十分重要。例如，当某些应用程序表现异常时，可以保证一致地正常运行和性能。

控制组机制始于2006年，内核从2.6.24版本开始被引入。

13.3　Linux 能力机制

Linux操作系统赋给普通用户尽可能低的权限，而把所有系统权限给予root用户。root用户可以执行一切特权操作。

事实上，那些需要root权限的程序往往只需要一种或几种特权操作，多数的特权操作都用不到。比如passwd程序只需要写passwd的权限，一个Web服务器只需要绑定到1024以下端口的权限。很显然，其他的特权对程序来说是不必要的，赋予程序root权限给系统带来了额外的威胁。如果这些程序有漏洞的话，那么理论上别人就可能利用漏洞取得系统的控制权，然后做他想做的任何事情。而如果把程序不必要的大多数特权去掉，那么即使存在漏洞，对我们造成的威胁也会小很多。

Linux的能力机制就是为了这个目的设计的。使用能力机制可以消除需要某些操作特权的程序对root用户的依赖，从而减小安全风险。系统管理员为了系统的安全，还可以去除root用户的某些能力，这样即使是root用户，也无法执行这些操作，而这个过程又是不可逆的。也就是说，如果一种能力被删除，除非重新启动系统，否则，即使root用户，也无法重新添加被删除的能力。

1. 能力的概念

Linux内核中使用的能力（capability）就是一个进程能够执行的某种操作。因为传统Linux系统中的root权限过于强大，能力机制把Linux的root权限细分成不同的能力，通过单独控制对每种能力的开关来达到安全的目的。这样的话，如果一个程序需要绑定低于1024的端口，那我们就赋予它这方面的能力，而不开放其他的各种能力。这样的话，如果程序的漏洞被利用，黑客也只能得到绑定低于1024的端口的能力，而不可能得到系统的控制权。

2. 能力边界集

能力机制还引入了能力边界集的概念。能力边界集是系统中所有进程允许拥有的能力的集合。如果在能力边界集中不存在某种能力，那么系统中所有进程都没有这种能力，即使以root权限运行也没有相应的能力。

删除系统中多余的能力对提高系统的安全性很有好处。假设你有一台重要的服务器，比较担心可加载内核模块的安全性，而你又不想完全禁止使用可加载内核模块，或者一些设备的驱动就是一些可加载内核模块。在这种情况下，最好使系统在启动时加载所有的模块，然后禁止加载/卸载任何内核模块。把CAP_SYS_MODULE从能力边界集中删除，系统将不再允许加载/卸载任何的内核模块。

3. 局限

虽然利用能力机制可以有效地保护系统安全，但是由于文件系统的制约（当前Linux文件结构没有存放能力机制的能力），Linux的能力机制还不是很完善。我们除了可以使用能力边界集从总体上放弃一些能力之外，还做不到只赋予某个程序某些方面的能力。

第 14 章 Docker API

在本章之前，我们对Docker的操作都是通过“docker+命令”这种方式，它是由Docker这个二进制文件加上命令来进行的。例如，列出本地所有镜像，可以使用docker images命令；获取当前正在运行的容器，可以使用docker ps命令等。本章要介绍另一种方法：Docker API，即Docker编程接口。本章内容主要包含：

- API概述，说明通过API操作Docker的优点以及API的分类等；
- 如何绑定Docker后台监听接口；
- 远程API，包括容器和镜像相关API；
- 平台API，先从机制原理上说明镜像的上传和下载流程，然后一步步说明操作步骤；
- API实战，通过docker-py库进行API实战编程。

14.1 API 概述

任何一个开放平台都会向开发者开放API，以供开发者更加自由地使用平台所提供的功能，定制出特定功能的应用。在应用中使用Docker API，应用就可以直接和Docker后台、库以及Docker Hub平台通信。事实上，Docker二进制本身也采用API与服务端通信。相对于使用“docker+命令”的方式对后台服务进行访问和使用，使用API方式具有以下几个优点。

- **无需安装Docker客户端。**采用“docker+命令”的方式，就意味着必须装有Docker客户端，而通过API方式则不需要。
- **效率更高。**除了在终端中直接使用“docker+命令”这种方式外，我们也可以在自己的应用中通过类似system(docker命令)的方式来达到同样的效果，然而和API编程相比，这种方式显得更为低效，因为它会新创建一个进程来执行system函数内部的操作。
- **更为自由。**“docker+命令”的形式返回的内容是经过Docker客户端处理的数据，并不是后台返回的原始数据，所以对有深度定制需求的应用而言，这制约了其自由定制的可能。

所以，要想更加高效和自由地使用Docker功能，就不得不了解Docker API编程。

从功能上分，Docker API包含三部分内容。

- Registry API（库API）：提供Docker库相关接口，用于保存镜像。
- Hub API（平台API）：和Docker Hub公共平台相关的接口。
- Remote API（远程API）：Docker客户端和后台服务端交互的接口。

在本章中，我们主要介绍远程API。因为相比库API和平台API，它更为核心和基础，拥有的操作也比其他两类多。此外，我们也会介绍与Docker Hub相关的API，因为它涉及镜像的上传和下载。本章中API提供的接口其实和前面章节涉及的Docker操作是一一对应的，所以在说明API用例时，读者肯定会有一种似曾相识的感觉，所以我们尽量以一种简单规范的形式来说明API，以一种“方法、用例请求、用例返回、参数、状态码、curl操作”模板的方法来展示API。当然，为了满足读者技痒需求，我们将会在14.5节中说明如何通过docker-py库来进行实战编程。

在介绍API的具体内容之前，我们需要说明一下API的工作原理。图14-1所示的是API中客户端和服务端的通信架构。图中包含两台机器——远端主机和Docker服务主机。Docker服务主机是一台装有Docker服务的计算机，它本身包含了Docker客户端和Docker后台。Docker后台提供Docker服务，外界通过套接字的方式对其进行访问。默认配置下，Docker后台只监听来自本地的通信请求，这主要是出于安全考虑。例如，图中的本地客户端就可以直接和后台套接字进行API通信。如果需要支持跨主机的API请求，我们需要在Docker后台启动时通过-H参数对指定或者全部网络接口进行绑定。

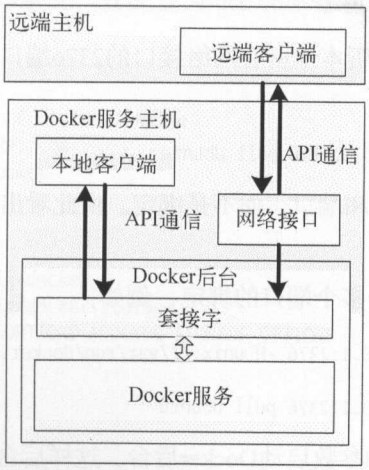


图14-1 API中客户端和服务端的通信架构

14.2 绑定 Docker 后台监听接口

在这一节中，我们来详细说明参数-H的用法。前面提到，Docker是由Docker后台和客户端组成的，而在默认配置下，Docker后台只接受来自本机的root用户的请求。事实上，Docker后台默认监听的是unix//var/run/docker.sock套接字文件，该文件位于/var/run/目录下，读者可以通过ls命

令来查看。那么，如果我想改变Docker后台的监听端口，甚至让其他主机也能够访问该后台服务，我该怎么做呢？

通过-H参数，可以让Docker后台监听指定的IP和端口。-H接受如下格式的IP和端口绑定：

```
tcp://[host][:port]
```

或

```
unix://path
```

在开始绑定之前，我们需要先把已经运行的Docker后台停止。如果是Ubuntu系列的系统，可以通过如下命令停止Docker后台：

```
$sudo service docker stop
```

如果是Red Hat系列，则为：

```
#systemctl stop docker.service
```

接下来，我们可以将本地所有网络接口的2376端口和Docker后端绑定。采用如下命令来启动Docker后台：

```
$ sudo docker -H 0.0.0.0:2376 -d &
```

Docker后台启动后，它将监听本地所有网络接口的2376端口，客户端可以通过-H参数来访问绑定后的端口，具体操作如下：

```
$ sudo docker -H tcp://127.0.0.1:2376 pull Ubuntu
```

此处的-H代表连接到指定IP和端口，而不是绑定。由此看出，-H参数在后台和客户端有着不一样的含义。

此外，Docker后台同时接受多个端口的绑定，例如：

```
$ sudo docker -H tcp://127.0.0.1:2376 -H unix:///var/run/docker.sock -d &
$ sudo docker search ubuntu
$ sudo docker -H tcp://127.0.0.1:2376 pull ubuntu
```

上面的代码通过两次使用-H参数启动Docker后台，这样后台不仅监听127.0.0.1:2376的请求，还监听/var/run/docker.sock上的请求。

除了使用命令行启动Docker后台外，一种更为持久化和自动化的方法是修改Docker服务的启动脚本，这免去了我们每次宕机之后手动关闭服务再启动的麻烦。这里我们仅说明Ubuntu系列和Red Hat系列如何操作。

1. Ubuntu和Debian系统

在Ubuntu或者Debian系统中，其配置为/etc/default/docker文件，打开该文件，其内容为：

```
# Docker Upstart and SysVinit configuration file
# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"
# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

将DOCKER_OPTS行改成：

```
DOCKER_OPTS="-H 0.0.0.0:2376  -H unix:///var/run/docker.sock"
```

保存后退出，然后重启服务使其生效。具体操作为：

```
$service docker stop
$service docker start
```

或者直接使用如下命令：

```
$service docker restart
```

2. Red Hat、Fedora和CentOS系统

Red Hat、Fedora和CentOS等系统，其配置为/usr/lib/system/system/docker.service文件，打开该文件，其内容为：

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target docker.socket
Requires=docker.socket
[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
ExecStart=/usr/bin/docker -d $OPTIONS $DOCKER_STORAGE_OPTIONS
LimitNOFILE=1048576
LimitNPROC=1048576
MountFlags=private
[Install]
WantedBy=multi-user.target
```

将下面这行代码：

```
ExecStart=/usr/bin/docker -d $OPTIONS $DOCKER_STORAGE_OPTIONS
```

改为：

```
ExecStart=/usr/bin/docker -d $OPTIONS $DOCKER_STORAGE_OPTIONS
-d --selinux-enabled -H tcp://0.0.0.0:2376  -H unix:///var/run/docker.sock
```

保存后退出，然后重启服务使其生效：

```
#systemctl restart docker.service
```

接下来，我们就可以学习Docker API了。

14.3　远程 API

在这一节中，我们主要介绍远程API，它主要包含对容器和镜像的操作。和“docker+命令”这种形式不一样，这里我们会一一说明常见的操作。为了便于演示，这里使用curl命令。它是一款利用URL语法在命令行下工作的开源文件传输工具，广泛应用于Unix、多种Linux发型版本中。当然，Windows系统下也有该工具的移植版本。

在第1章中，我们知道通过docker info命令可以查询当前Docker的系统信息。下面我们使用API方法来获得同样的信息，其操作如下：

```
$ curl -X GET http://localhost:2376/info
{"Containers":4,"Debug":0,"Driver":"aufs","DriverStatus":[["Root Dir","/var/lib/docker/aufs"],
["Dirs","61"]],"ExecutionDriver":"native-0.2","IPv4Forwarding":1,"Images":53,"IndexServerAddress":
    "https://index.docker.io/v1/","InitPath":"/usr/bin/docker","InitSha1":"","KernelVersion":"3.13.0-44-
    generic","MemoryLimit":1,"NEventsListener":0,"NFd":10,"NGoroutines":10,"OperatingSystem":"Ubuntu
    14.04.1 LTS","SwapLimit":0}
```

我们使用curl工具来请求http://locahost:2376/info，该路径用于获取Docker系统信息，-X参数后面接想要执行的HTTP操作，这里是GET。可以看到，返回的信息与docker info命令一样。

查看Docker的版本信息，可以通过如下代码实现：

```
$ curl -X GET http://localhost:2376/version
{"ApiVersion":"1.16","Arch":"amd64","GitCommit":"5bc2ff8","GoVersion":"go1.3.3","KernelVersion":"3
.13.0-24-generic","Os":"linux","Version":"1.4.1"}
$ docker version
Client version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): 5bc2ff8
OS/Arch (client): linux/amd64
Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3
Git commit (server): 5bc2ff8
```

可以看到，通过API方式和docker version命令获得的信息是一样的。

接下来，我们分别说明容器和镜像相关的API。

14.3.1　容器相关的 API

容器相关的API都在/containers/路径下面。接下来，我们分别说明容器的操作。由于通过curl命令来访问API的操作相对简单，操作本身无需再重复演示说明，所以接下来就将焦点集中在API

本身。根据API的规范性，我们会按方法、用例请求、用例返回、请求参数、返回的状态码以及curl操作这种标准模板的形式来逐个说明，这样不仅可以满足读者动手操作的需求，而且可以将本部分内容作为手册翻阅。

1. 列出容器

方法： GET /containers/json

用例请求： GET /containers/json?all=1&before=8dfafdbc3a40&size=1 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
Content-Type: application/json
[
    {
        "Id": "8dfafdbc3a40",
        "Image": "base:latest",
        "Command": "echo 1",
        "Created": 1367854155,
        "Status": "Exit 0",
        "Ports":[{"PrivatePort": 2222, "PublicPort": 3333, "Type": "tcp"}],
        "SizeRw":12288,
        "SizeRootFs":0
    },
    ...
]
```

请求参数

- all：表示是否显示所有容器，其值为1/True/true或0/False/false。若为真，则会显示全部（包含已经停止的）容器，否则只显示正在运行的容器，默认值为0。
- limit：仅显示最新建立的几个容器。
- since：显示比指定Id的容器更晚创建的容器。
- before：显示比指定Id的容器更早创建的容器。
- size：是否显示容器的大小，其值为1/True/true或0/False/false。
- filters：使用JSON格式的条件过滤容器，例如退出码（exited=<int>），status运行状态（status=restarting|running|paused|exited）。

返回的状态码

- 200：返回正常。
- 400：参数错误。
- 500：服务器错误。

curl操作：

```
curl -X GET http://localhost:2376/containers/json?all=1
```

curl命令中的-X参数后面跟要执行的HTTP操作，这里是GET操作，后续我们还会看到POST、PUT、DELETE操作。

2. 创建容器

方法： POST /containers/create

用例请求：

```
POST /containers/create HTTP/1.1
Content-Type: application/json
{
    "Hostname":"",
    "Domainname": "",
    "User":"",
    "Memory":0,
    "MemorySwap":0,
    "CpuShares": 512,
    "Cpuset": "0,1",
    "AttachStdin":false,
    "AttachStdout":true,
    "AttachStderr":true,
    "Tty":false,
    "OpenStdin":false,
    "StdinOnce":false,
    "Env":null,
    "Cmd":[
        "date"
    ],
    "Entrypoint": "",
    "Image":"base",
    "Volumes":{
        "/tmp": {}
    },
    "WorkingDir":"",
    "NetworkDisabled": false,
    "MacAddress":"12:34:56:78:9a:bc",
    "ExposedPorts":{
        "22/tcp": {}
    },
    "SecurityOpts": [""],
    "HostConfig": {
        "Binds":["/tmp:/tmp"],
        "Links":["redis3:redis"],
        "LxcConf":{"lxc.utsname":"docker"},
        "PortBindings":{ "22/tcp": [{ "HostPort": "11022" }] },
        "PublishAllPorts":false,
        "Privileged":false,
        "Dns": ["8.8.8.8"],
        "DnsSearch": [""],
        "VolumesFrom": ["parent", "other:ro"],
        "CapAdd": ["NET_ADMIN"],
        "CapDrop": ["MKNOD"],
```

```
        "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
        "NetworkMode": "bridge",
        "Devices": []
    }
}
```

用例返回：

```
HTTP/1.1 201 Created
Content-Type: application/json
{
    "Id":"e90e34656806"
    "Warnings":[]
}
```

JSON参数

- Hostname：容器内系统的主机名。
- Domainname：容器内系统的域名。
- User：容器内用户。
- Memory：内存（字节）。
- MemorySwap：包含Swap交换存储在内的总存储量。
- AttachStdin：是否附加到标准输入。
- AttachStdout：是否附加到标准输出。
- AttachStderr：是否附加到标准错误输出。
- Tty：是否需要附加伪终端。
- Env：环境变量，形如VAR=value。
- Cmd：容器要运行的命令。
- Entrypoint：容器的入口点。
- Image：容器基于的镜像。
- Volumes：数据卷。
- WorkingDir：默认工作目录。
- NetworkDisabled：是否打开网络。
- ExposedPorts：暴露端口，形如 "ExposedPorts": { "<port>/<tcp|udp>: {}" }。
- SecurityOpts：安全选项，例如配置SELinux。
- HostConfig：子项配置。
- Binds：数据卷配置，形如host_path:container_path:ro。
- Links：容器连接，形如container_name:alias。
- PortBindings：端口映射，形如{ <port>/<protocol>: [{ "HostPort": "<port>" }] }。
- PublishAllPorts：是否开放所有服务端口，并随机映射到本地主机。
- Privileged：是否给予容器root权限访问宿主主机。

- Dns：Dns列表。
- DnsSearch：Dsn搜索域。
- VolumesFrom：数据容器引用，形如[:]`。
- CapAdd：容器所使用的内核机制列表。
- Capdrop：容器不适用的内核机制列表。
- RestartPolicy：容器退出时的重启机制。
- NetworkMode：容器的网络类型，支持bridge、host和container:<name|id>。

请求参数

- name：容器的名字。

状态码

- 201：返回正常。
- 404：无该容器。
- 406：容器不能附加为终端（容器已经停止）。
- 500：服务器错误。

curl操作：

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:2376/containers/create -d '{
    "Hostname":"",
    ...
}'
```

curl命令执行了POST请求，参数-H表示需要添加的HTTP头，这里添加了Content-Type: application/json表明应用的数据类型为JSON。参数-d后面跟的是JSON数据，这是POST投递的数据体，用单引号括起。这里我们把创建容器的JSON配置投递给Docker后台。

3. 启动容器

方法： POST /containers/(id)/start

用例请求：

```
POST /containers/(id)/start HTTP/1.1
Content-Type: application/json
```

用例返回： HTTP/1.1 204 No Content

参数：（无）

状态码：（略）

curl操作： curl -X POST http://localhost:2376/containers/4fa6e0f0c678/start

创建好容器之后，我们可以通过start操作来启动该容器。在上述curl操作中，containers/后面是容器的Id，无需写全，可以使用Id的前面几位，即能够区分不同容器就行了。

4. 停止容器

方法： POST /containers/(id)/stop

用例请求： POST /containers/e90e34656806/stop?t=5 HTTP/1.1

用例返回： HTTP/1.1 204 No Content

参数

- t：延时多少秒后停止。

状态码

- 304：容器已经停止。

curl操作： curl -X POST http://localhost:2376/containers/4fa6e0f0c678/stop

5. 删除容器

方法： DELETE /containers/(id)

用例请求： DELETE /containers/16253994b7c4?v=1 HTTP/1.1

用例返回： HTTP/1.1 204 No Content

参数

- v：其值为1/True/true或者0/False/false，用于删除关联的数据卷，默认值为false。
- force：其值为1/True/true或者0/False/false，用于强制删除，默认值为false。

状态码：（略）

curl操作： curl -X DELETE http://localhost:2376/containers/4fa6e0f0c678

删除容器发送的HTTP操作是DELETE。

关于容器API的示例我们就说到这里。为了方便读者快速查阅，我们将容器相关的API总结一下，具体如表14-1所示。

表14-1 常用容器相关API列表

功 能	方 法	参 数
列出容器	GET /containers/json	all、limit等
创建容器	POST /containers/create	name、JSON参数
查看信息	GET /containers/(id)/json	
正在运行的进程	GET /containers/(id)/top	
获取日志	GET /containers/(id)/logs	follow、stdout、stderr等
查看文件变更	GET /containers/(id)/changes	
导出	GET /containers/(id)/export	

（续）

功　能	方　法	参　数
启动	POST /containers/(id)/start	
停止	POST /containers/(id)/stop	t
重启	POST /containers/(id)/restart	t
杀死	POST /containers/(id)/kill	signal
附加终端	POST /containers/(id)/attach	logs、stdin、stdout等
暂停	POST /containers/(id)/pause	
重新运行暂停的容器	POST /containers/(id)/unpause	
等待容器停止	POST /containers/(id)/wait	
删除	DELETE /containers/(id)	v、force
从容器复制目录/文件	POST /containers/(id)/copy	

更多关于容器API的详情，请参见B.1节。

14.3.2　镜像相关的 API

和镜像相关的操作主要有列出当前主机的所有镜像、创建镜像、查看镜像信息、查看镜像历史、将镜像推送入库、给镜像添加标签、删除和搜索镜像等。所有和镜像相关的API，其访问路径都在/images/之下。由于镜像的上传、下载和搜索都要和Docker Hub打交道，我们会在14.4节介绍完Docker Hub的架构和流程之后学习它们。

1. 列出镜像

方法： GET /images/json

用例请求： GET /images/json?all=0 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
Content-Type: application/json
[
    {
        "RepoTags": [
            "ubuntu:12.04",
            "ubuntu:precise",
            "ubuntu:latest"
        ],
        "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
        "Created": 1365714795,
        "Size": 131506275,
        "VirtualSize": 131506275
    },
    ...
]
```

参数

- all：其值为1/True/true或者0/False/false，表示是否返回所有镜像，默认值为false。
- filters：JSON形式的条件过滤。

状态码：（略）

curl操作：curl -X POST http://localhost:2376/ images/json?all=0

2. 查看镜像详细信息

方法：GET /images/(name)/json

用例请求：GET /images/base/json HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "Created":"2013-03-23T22:24:18.818426-07:00",
    "Container":"3d67245a8d72ecf13f33dffac9f79dcdf70f75acb84d308770391510e0c23ad0",
    "ContainerConfig":
    {
        "Hostname":"",
        "User":"",
        "Memory":0,
        "MemorySwap":0,
        "AttachStdin":false,
        "AttachStdout":false,
        "AttachStderr":false,
        "PortSpecs":null,
        "Tty":true,
        "OpenStdin":true,
        "StdinOnce":false,
        "Env":null,
        "Cmd": ["/bin/bash"],
        "Dns":null,
        "Image":"base",
        "Volumes":null,
        "VolumesFrom":"",
        "WorkingDir":""
    },
    "Id":"b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Parent":"27cf784147099545",
    "Size": 6824592
}
```

参数：（无）

状态码

- 404：无该镜像。

curl操作：

```
curl -X GET http://localhost:2376/ images/base/json
```

3. 创建镜像

创建镜像有3种方法：从Docker Hub等仓库下载，通过Dockerfile制作以及将容器提交为镜像。关于通过从仓库下载来创建镜像，我们将会在14.4.2节中说明，这里我们主要说明如何通过Dockerfile创建镜像和将容器提交成镜像。

使用Dockerfile创建镜像

首先，我们编写一个Dockerfile文件，然后将其压缩，最后使用API build创建镜像。具体内容如下：

```
$ cat Dockerfile
From Ubuntu
RUN echo  “deb http://archive.ubuntu.com/ubuntu precise main univers”  > /etc/apt/ sources.list
mkdir /root
RUN apt-get update
EXPOSE 22 80
CMD echo  “Using Dockerfile to build image now !”
$tar zcf Dockerfile.tar.gz Dockerfile
Curl -v -X POST -H  “Content-Type: application/tar”  --data-binary  ‘@Dockerfile.tar.gz’
http://localhost:2376/build?t=testbuild
```

参数HTTP头中的Content-Type: application/tar字段表明POST方法投递的是一个压缩包。--data-binary则说明投递的是一个二进制数据。和--data-binary相对应的是--data-ascii，它表示投递的数据以ASCII编码。参数t=testbuild表示所创建的镜像名为testbuild。

将容器提交成镜像

在3.3节中，我们知道使用docker commit命令可以将一个容器提交成镜像。现在，我们通过API来实现同样的操作，具体操作如下：

```
$curl -X POST -H "Content-Type: application/json" http://localhost:2376/commit? container=ae17d&
    comment=test&repo=myrepo
```

这样，我们就将Id为ae17d的容器提交为名为myrepo的镜像。

关于镜像的API示例，现在就说到这里。和容器一样，为了方便读者查阅，我们将镜像相关的API归纳为表14-2所示。

表14-2　镜像相关的API

功　能	方　法	参　数
列出镜像	GET /images/json	all、filters
创建（下载创建）镜像	POST /images/create	fromImage、repo、tag等
查看镜像信息	GET /images/(name)/json	

（续）

功　能	方　法	参　数
获取镜像历史	GET /images/(name)/history	
推送镜像到仓库	POST /images/(name)/push	
给镜像贴标签	POST /images/(name)/tag	repo、force、tag
删除镜像	DELETE /images/(name)	force
搜索镜像	GET /images/search	Term

更多关于镜像API的详细说明，可以参见附录B.2。

14.4 平台 API

本节主要介绍Docker Hub的API，但在这之前，我们有必要说明Docker Hub的组成结构以及常见操作的交互流程。

14.4.1 注册服务器架构及流程

Docker的注册服务器，也称Docker Registry，它是Docker的重要组件之一。Docker Registry包含3个角色，分别是Index（索引）、Registry（库）和Registry客户端。

- Index：负责维护用户账号、镜像校验以及公共命名空间这些信息。它通过Web UI、元数据存储、认证服务和服务化等组件来维护这些信息。
- Registry：它是镜像和图标的仓库，由S3、云文件和本地文件系统来提供数据库支持。常见的Registry有Sponsor Registry、Mirror Registry、Vendor Registry和Private Registry。
- Registry客户端：用户通过客户端来与Registry通信、鉴权、推送和拉取信息。

这三个组件是如何分工合作的呢？接下来，我们以Docker Hub为例，通过常见的拉取、推送镜像的流程来说明一下。

1. 下载镜像

用户通过客户端向注册服务器下载一个指定名字的镜像，其流程如图14-2所示，具体步骤如下所示。

(1) 客户端向Index发送下载某个镜像的请求。

(2) Index向客户端返回三部分信息：该镜像位于Registry库的位置、该镜像包括所有层的校验信息、授权token。当第(1)步中的请求头有X-Docker-Token时，才会返回token，它对于私人仓库是需要的，而对于公有库则不是必要的。

(3) 用户通过镜像的位置以及授权token向Registry发出请求。

(4) Registry向Index核实该token是否被授权。

(5) Index返回授权验证结果，如果合法，则Registry通知客户端可以下载该镜像，否则拒绝此

次下载请求。

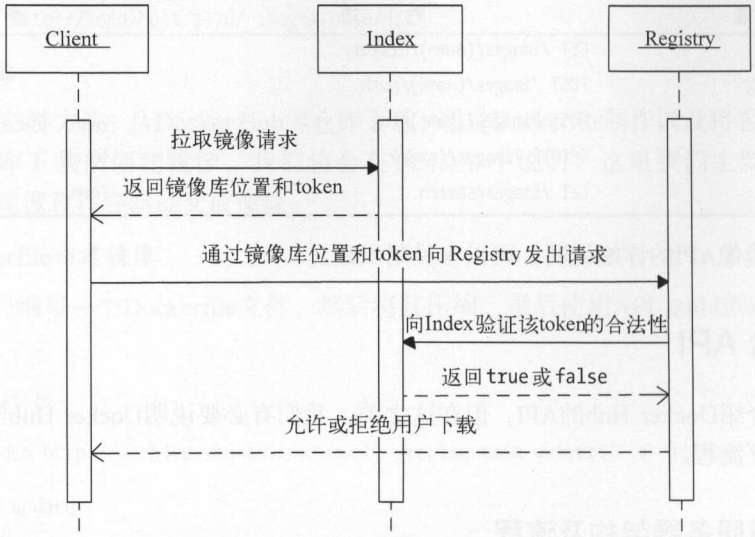


图14-2　客户端向注册服务器拉取镜像流程

2. 上传镜像

用户通过客户端向注册服务器上传镜像的流程如图14-3所示，具体步骤如下。

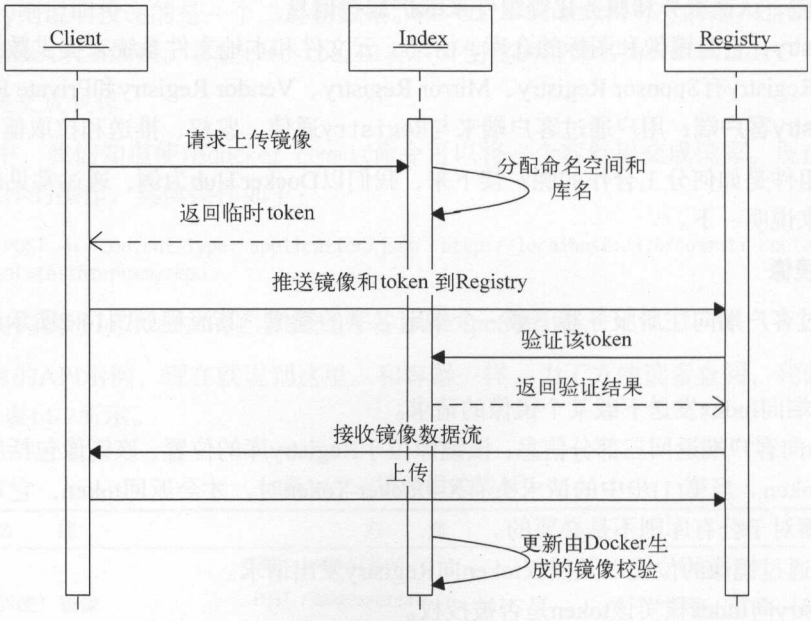


图14-3　客户端向注册服务器上传镜像的流程

(1) 用户发送带证书的请求到Index，要求分配库名。

(2) 在成功认证以及确定命名空间、库名都可以分配之后，Index向客户端返回一个临时的token。

(3) 镜像连带临时token被推送到Registry。

(4) Registry向Index验证该token的有效性，认证成功后开始读取来自客户端的数据流，否则拒绝该次上传请求。

(5) Index更新此次镜像的校验信息。

介绍完Docker Hub的组件和工作流程之后，接下来开始介绍Hub API。

14.4.2 操作 Hub API

在本节中，我们主要介绍如何在Docker Hub上操作API。首先介绍用户如何注册和登录，然后说明如何通过API在Docker Hub上新建仓库，更新仓库，查看仓库中已有的镜像以及删除仓库。最后，将介绍如何上传镜像到Docker Hub的库中以及如何下载镜像。

1. 用户注册

注册新用户需要设置邮箱、用户名和密码。

方法： `POST /v1/users/`

用例请求：

```
POST /v1/users/ HTTP/1.1
    Host: index.docker.io
    Accept: application/json
    Content-Type: application/json
    {"email": "sam@docker.com",
        "password": "toto42",
        "username": "foobar"}
```

用例返回：

```
HTTP/1.1 201 OK
    Vary: Accept
    Content-Type: application/json
"User Created"
```

JSON参数

- email：电子邮箱，注册后需要去邮箱激活账号。
- username：用户名，最少4个字节，最大30个字节，必须由[a-z0-9_]组成。
- password：密码，最短为5个字节。

curl操作：

```
$ curl -X POST -v -H "Content-Type: application/json" https://index.docker.io/v1/users/ -d
'{"email":"zengjinlong@xunlei.com","password":"110110","username":"zengjinlong123"}'
```

2. 用户登录

注册完用户并激活之后，我们可以用该账号登录Docker Hub了。

方法：GET /v1/users/

用例请求：

```
GET /v1/users/ HTTP/1.1
    Host: index.docker.io
    Accept: application/json
    Authorization: Basic akmklmasadalkm==
```

用例返回：

```
HTTP/1.1 200 OK
    Vary: Accept
    Content-Type: application/json
    OK
```

状态码

- 200：返回正确。
- 401：认证失败。
- 403：账号尚未激活。

curl操作

在HTTP 头的Authorization字段中，后面的值是Basic加“用户名:密码”的Base64编码。例如，我的用户名为helloworld，密码为123456，则要先将其进行Base64加密，具体操作如下：

```
$ USER=`echo 'hellworld:12345' | base64 --wrap=0`
$ echo $USER
aGVsbHdvcmxkOjEyMzQ1Cg==
$ curl -v -X GET -H "Authorization: Basic $USER" https://index.docker.io/v1/users/
```

另一种不需要生成Base64的方法是使用--user参数，具体操作如下：

```
$ curl -v -X GET --user hellworld:123456 https://index.docker.io/v1/users/
```

如果登录成功，会返回200，否则返回401表示认证失败。

我们推荐使用第二种方法，因为它比较直观。事实上，第二种方法和第一种方法本质上一样。--user字段最终也会被翻译成Authorization字段，读者可以通过curl的-v参数查看具体的交互过程。

3. 新建仓库

用户可以通过API的形式在Docker Hub上创建镜像仓库，此时需要提供命名空间和仓库名。一般情况下，命名空间为用户名。

方法： PUT /v1/repositories/(namespace)/(repo_name)/

用例请求：

```
PUT /v1/repositories/foo/bar/ HTTP/1.1
    Host: index.docker.io
    Accept: application/json
    Content-Type: application/json
    Authorization: Basic akmklmasadalkm==
    X-Docker-Token: true
    [{"id": "9e89cc6f0bc3c38722009fe6857087b486531f9a779a0c17e3ed29dae8f12c4f"}]
```

用例返回：

```
HTTP/1.1 200
    Vary: Accept
    Content-Type: application/json
    WWW-Authenticate: Token signature=123abc,repository="foo/bar",access=write
    X-Docker-Token: signature=123abc,repository="foo/bar",access=write
    X-Docker-Endpoints: registry-1.docker.io [, registry-2.docker.io]
    ""
```

参数：（无）

状态码

- 200：创建成功。
- 400：错误，通常为JSON内部格式或者参数错误。
- 401：认证失败。
- 403：账户尚未激活。

curl操作：

```
$ curl -v -L -X PUT --user helloworld:123456 -H "Accept: application/json" -H "Content-Type:
    application/json"  --post301 https://index.docker.io/v1/repositories/helloworld/foo2 -d '[{"id":
    "1cb837a9709a2fff9591c4ca4ff6f336b0c98308ac1bbb493179888787a6c691"}]'
```

JSON里面添加的是加入该仓库的镜像的id列表。

4. 上传镜像

接下来，我们需要说明如何把一个镜像上传到Docker Hub中。这分为两部分，首先需要将本地的镜像标记成Docker Hub响应库里的镜像，然后通过push操作把镜像从本地上传到Docker Hub中。

本地现有镜像如下：

```
micall@micall-ThinkPad:~$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
dockerfile/redis        latest              1cb837a9709a        2 weeks ago
progrium/ambassadord    latest              ebc261f84aa4        3 weeks ago         224.4 MB
ubuntu                  latest              86ce37374f40        7 weeks ago         192.7 MB
minimicall/node_web     0.1                 730770dff17f        7 weeks ago         268.7 MB
```

我们要把dockerfile/redis上传到helloworld用户的foo仓库，所以需要先把该镜像签入helloworld/foo。具体操作为：

```
curl -v -X POST http://localhost:2376/images/dockerfile/redis/tag?repo=minimicall/foo&tag=lat est
```

标记完之后，我们可以看到minimicall/foo镜像，它和dockerfile/redis一模一样，具体为：

```
micall@micall-ThinkPad:~$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
dockerfile/redis        latest              1cb837a9709a        2 weeks ago         419.1 MB
minimicall/foo          latest              1cb837a9709a        2 weeks ago         419.1 MB
```

好了，终于到了上传的一刻！其操作为：

```
~$ XAR=`echo '{"username":"minimicall","password":"110110","email":"470910357@qq.com",
"serveraddress":"index.docker.io"}' | base64 --wrap=0`
$echo $XAR
eyJ1c2VybmFtZSI6Im1pbmltaWNhbGwiLCJwYXNzd29yZCI6IjExMDExMCIsImVtYWlsIjoiNDcwOTEwMzU3QHFxLmNvbSIsIn
NlcnZlcmFkZHJlc3MiOiJpbmRleC5kb2NrZXIuaW8ifQo=
    $ curl -v -X POST -L --post301 -H "X-Registry-Auth:$XAR" http://localhost:2376/images/minimicall
    /node_web/push
```

在上述操作中，我们首先通过用户名、密码、邮箱和服务器地址的JSON格式来生成Base64编码，然后赋给变量XAR。上传操作是images下的push操作，它需要加入HTTP头X-Registry-Auth字段，其值为XAR的值。上传过程需要等待，根据镜像的大小和网络状况决定。至此，我们成功地使用API将镜像上传到Docker Hub中。

5. 下载镜像

下载镜像是创建镜像的一种方式。

方法： POST /images/create

用例请求： POST /images/create?fromImage=base HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {"status":"Pulling..."}
    {"status":"Pulling", "progress":"1 B/ 100 B", "progressDetail":{"current":1, "total":100}}
    {"error":"Invalid..."}
    ...
```

参数

- fromImage：基础镜像。
- fromSrc：导入源，一个可以获得镜像的URL地址。
- repo：仓库。
- tag：标签。

- registry：拉取镜像的注册服务器。

curl操作：

```
$ curl -v -X POST http://localhost:2376/images/create?fromImage=minimicall/foo&tag=latest
   [1] 22386
micall@micall-ThinkPad:~$ * Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 2376 (#0)
> POST /images/create?fromImage=minimicall/foo HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:2376
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Fri, 16 Jan 2015 02:48:42 GMT
< Transfer-Encoding: chunked
<
{"status":"Pulling repository minimicall/foo"}
{"status":"Status: Image is up to date for minimicall/foo"}
```

带上-v参数的curl操作输出了HTTP交互的整个过程。由于此镜像是由本地上传上去的，所以无需再次下载。最后一行代码显示已经更新到最新了。

14.5 API 实战：docker-py 库编程

在前面几节中，我们使用curl命令来操作API接口，本节中我们将会使用Docker官方提供的docker-py库来进行API编程。docker-py是Docker官方提供的Python版本的API接口库。下面我会先构建其开发环境，然后通过几个实例说明它的用法。

14.5.1 docker-py 开发环境的搭建

docker-py的开发环境包含Python环境和docker-py库的配置，系统环境是Ubuntu 14.04。

1. Python环境的搭建

Ubuntu 14.04自带有Python开发环境，可以通过如下命令查看其版本号：

```
# python --version
Python 2.7.6
```

如果你的系统没有Python开发环境，则需要自己安装。需要注意的是，Python 3和Python 2不兼容。Python及其包管理工具pip的安装如下：

```
$ sudo apt-get install python-dev
$ sudo apt-get install libevent-dev
$ sudo apt-get install python-pip
```

2. 安装docker-py

安装docker-py库，只需要从GitHub中拉取其压缩包，在本地解压，然后通过Python脚本安装即可，具体操作为：

```
# wget https://github.com/docker/docker-py/archive/master.zip
# unzip master
# cd docker-py-master/
# python setup.py install
```

3. 验证是否安装成功

通过如下命令可以验证docker-py是否安装成功：

```
# python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import docker
>>>
```

如果import docker没有提示出错，就说明docker-py已经正确安装了。

14.5.2　docker-py 库编程

开发环境已经搭建完毕，接下来进入编程环节。

1. 连接Docker后台

在14.2节中，我们已经知道连接后台有两种方式，一种是tcp，一种是unix socket方式。要管理Docker，必然先要连接到Docker后台。docker-py库支持这两种方式连接Docker后台，下面我们通过一个例子来说明其用法。首先，用Vim创建一个doc_info.py文件，其内容为：

```
import docker
cli = docker.Client(base_url='tcp://127.0.0.1:2376')
#cli = docker.Client(base_url='unix://var/run/docker.sock')
print cli.info()
```

第一行用import指令引入了docker包，然后对Docker后台进行连接。读者可以选取两种方式中的一种，tcp方式可以连接其他主机的Docker后台。Client函数的参数如下所示。

- base_url：指定要连接到的Docker后台，包含IP地址和监听的端口。
- version：API的版本号。
- timeout：超时时间。
- tls：tls支持。

这里我们只设置了base_url参数，其他的都采用默认参数。在上述代码中，读者可以将注释部分打开，对比两种不同的连接方式。

最后，使用info函数输出Docker的相关信息。保存该文件，然后在命令行下运行python dock_info.py，具体如下：

```
$ python docker_info.py
{u'NGoroutines': 77, u'DockerRootDir': u'/var/lib/docker', u'DriverStatus': [[u'Root Dir', u'/var/
    lib/docker/aufs'], [u'Dirs', u'387']], u'OperatingSystem': u'Ubuntu 14.04 LTS', u'Containers':
    64, u'MemTotal': 8181542912, u'Driver': u'aufs', u'IndexServerAddress':
    u'https://index.docker.io/v1/', u'InitPath': u'/usr/bin/docker', u'ExecutionDriver':
    u'native-0.2', u'Name': u'micall-ThinkPad', u'NCPU': 4, u'Debug': 0, u'ID':
    u'3MDX:LBWZ:TA5F:KR2T:LI44:34VF:PDLQ:5ZDF:3BOQ:PEPS:OMIP:DTWU', u'IPv4Forwarding': 1,
    u'KernelVersion': u'3.13.0-24-generic', u'NFd': 68, u'InitSha1': u'', u'Labels': None,
    u'MemoryLimit':1, u'SwapLimit': 0, u'Images': 255, u'NEventsListener': 2}
```

可以看到，所获取的信息和使用docker info命令获得的信息一致。

2. 查看本地镜像

接下来，我们可以查看本地所有的镜像。创建文件doc_images.py，其内容为：

```
import docker
cli = docker.Client('tcp://127.0.0.1:2376')
print cli.images()
```

运行python doc_images.py命令：

```
# python doc_images.py
[{u'Created': 1422350386, u'VirtualSize': 889663485, u'ParentId':
    u'1d1ee8ba08f761fddc3c40b8baad7c3958b34f3a35bb900516763adb5a5cd1b7', u'RepoTags':
    [u'minimicall/centos-mysql:v1'], u'Id':
    u'a34a95bedc997f382b2662cf77f230448f43599fd09b9c7fd14bd1b84b2c36f5', u'Size': 0}, {u'Created':
    1421911413, u'VirtualSize': 744788608, u'ParentId':
    u'5e927153cfba10fadef255b55c92dd623f43dbd33444c0f
    ...
```

3. 构建容器

我们也可以直接在Python命令环境中编程，此时直接在命令行下敲入python即可。下面我们使用Dockerfile构建一个容器：

```
>>> from io import BytesIO
>>> from docker import Client
>>> dockerfile = '''
... #Shared Volume
... FROM busybox:buildroot-2014.02
... MAINTAINER minimicall, 470910357@qq.com
... VOLUME /data
... CMD ["/bin/sh"]
... '''
>>> f = BytesIO(dockerfile.encode('utf-8'))
>>> cli = Client(base_url='tcp://127.0.0.1:2376')
>>> response = [line for line in cli.build(fileobj=f,rm=True,tag='minimicall/volume')]
>>>
>>> response
```

```
['{"stream":"Step 0 : FROM busybox:buildroot-2014.02\\n"}\r\n', '{"status":"The image you are pulling
has been verified","id":"busybox:buildroot-2014.02"}\r\n', '{"status":"Pulling fs
layer","progressDetail":{},"id":"ea13149945cb"}'
```

可以看到，我们基于busybox构建了一个名为minimicall/volume的镜像。创建完之后，可以查看是否存在该容器：

```
>>> cli.images()
[{u'Created': 1422772698, u'VirtualSize': 2433303, u'ParentId':
u'ff311d5495905d69b5cd603825fc4cd76078ed9a6fe064edc84f4247950c74c8', u'RepoTags':
[u'minimicall/volume:latest'], u'Id':
u'99dbb68a761cda72afb4e469c6c1bec6dabf4a88adde2fd7e9b236abdcd900b6', u'Size': 0},
```

可见，minimicall/volume:latest已经成功创建了。

4. 启动容器

有了镜像，接下来以该镜像为基础启动容器：

```
>>> container = cli.create_container(image='busybox:latest',command='/bin/sleep 30')
>>> print container
{u'Id': u'4d21ff66e440f989f0a24c6c29990e121000aa198385326a768c97a977f56650', u'Warnings': None}
>>>
```

docker-py的API函数都是根据Docker API编写的，所以这里就不再列举更多的例子了。更多关于docker-py的API编程的内容，可以访问其官网：https://github.com/docker/docker-py。

第 15 章 Fig

复杂的应用往往包含多个组件，其中每个组件都应该安装到独立的容器内部，但是采用手工方式管理容器显得非常麻烦，所以在这一章中，我们将介绍一款多容器管理的利器——Fig。本章内容主要包括：

- Fig简介
- Fig安装
- Rails开发环境配置
- Django开发环境配置
- WordPress开发环境配置
- Flocker——跨主机的Fig应用

15.1 Fig 简介

一开始学习Docker时，一般我们只需要操作一个容器。然而有些应用需要多个服务来协作提供，这时把这些服务都安装到一个容器内显然不太合适，并且这与我们使用Docker的初衷背道而驰，因为它没有提供良好的隔离性。更好的方法是使用多个容器来分别安装它们，然后协调管理这些容器。而当容器的数量增多时，手动管理容器之间的连接、数据卷等配置会越来越力不从心，此时需要一种工具来替代手动管理，将我们从繁杂的基础管理中解脱出来，而Fig就是这么一款工具。

Fig是一款基于Docker的用于快速搭建开发环境的工具，它由Orchard团队开发并遵循Apache 2.0协议。目前，该团队已经加入Docker团队中。Fig通过一个YAML配置文件来管理多个Docker容器，非常适合组合使用多个容器进行开发的场景。

15.2 Fig 安装

安装Fig时，要保证Docker的版本号不得低于1.3。目前，Fig仅支持Linux和OS X系统。Fig所有的版本都会在https://github.com/docker/fig/releases上更新，目前最新版本是1.0.1。既可以通过浏览器直接下载二进制文件，然后进行安装，也可以通过如下命令进行安装：

```
curl -L https://github.com/docker/fig/releases/download/1.0.1/fig-`uname -s`-`uname -m` >
/usr/local/bin/fig; chmod +x /usr/local/bin/fig
```

上述命令首先通过curl下载Fig二进制文件，然后将其复制到/usr/local/bin目录下，最后更改其权限使其可以运行。

安装完毕后，可以通过如下命令来验证安装是否正确：

```
$fig version
fig 1.0.1
```

15.3　Rails 开发环境配置

在这一节中，我们将使用Fig来配置Rails+PostgreSQL应用。

首先，新建一个目录，并为其创建3个文件。第一个是Dockerfile文件，用于构建Web服务镜像，其内容为：

```
FROM ruby
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
RUN mkdir /myapp
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
RUN bundle install
ADD . /myapp
```

该Dockerfile文件构建的Docker将会包含Ruby、Bundle、代码以及依赖的所有组件。

然后，编写一个Gemfile文件来加载Rails，它将会被rails new改写。其内容如下：

```
source 'https://rubygems.org'
gem 'rails', '4.2.0'
```

文件中rails的版本号4.2.0是当前最新版本。如果安装过程中出现错误，读者应该到rails官网查看最新的版本号。

最后，定义一个fig.yml文件，其内容包括：应用由哪些组件组成，例如由一个数据库加Web应用组成；每一个服务的镜像是如何获得的；容器之间是如何互联的，暴露哪些网络端口等。其内容为：

```
db:
    image: postgres
    ports:
        - "5432"

web:
    build: .
    command: bundle exec rackup -p 3000
    volumes:
        - .:/myapp
```

```
    ports:
        - "3000:3000"

    links:
        - db
```

配置定义了db和web两个服务。db采用postgres镜像，对外暴露5432端口。web服务则通过当前目录下的Dockerfile来构建，Dockerfile即为上面所讲到的Dockerfile。command定义了执行的命令，volumes将主机的当前目录映射为容器内部的/myapp数据卷。ports将主机的3000端口和容器的3000端口映射。links定义了web到db容器的连接。

接下来，使用fig run命令来生成一套Rails项目骨架：

```
$ fig run web rails new . --force --database=postgresql --skip-bundle
```

执行该命令会耗费比较长的时间，这取决于网络速度，因为需要去Docker Hub拉取相关镜像。Fig首先会去构建Web服务镜像，然后使用该镜像生成一个容器并在里面执行rails new。

命令执行完之后，当前目录的内容会发生变化，具体如下：

```
$ ls
Dockerfile   app          fig.yml      tmp
Gemfile      bin          lib          vendor
Gemfile.lock config       log
README.rdoc  config.ru    public
Rakefile     db           test
```

编辑Gemfile文件，去掉加载therubyracer行的注释，我们就获得一个JavaScript运行环境：

```
gem 'therubyracer', platforms: :ruby
```

我们使用fig build命令再构建一次镜像：

```
$ fig build
```

运行fig build命令之后，我们就得到了两个可以立即运行的容器，分别为db和web。默认情况下，Rails是使用localhost上的数据库，这里需要将它重定向到db容器，并修改相关数据库连接配置。打开新生成的database.yml文件，将其内容替换为如下内容：

```
development: &default
    adapter: postgresql
    encoding: unicode
    database: postgres
    pool: 5
    username: postgres
    password:
    host: db
test:
    <<: *default
    database: myapp_test
```

现在，我们可以运行整个应用程序了，操作为：

```
$ fig up
```

如果一切运行正常，你将会看到一些PostgreSQL的日志输出，具体如下：

```
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick 1.3.1
myapp_web_1 | [2014-01-17 17:16:29] INFO  ruby 2.0.0 (2013-11-22) [x86_64-linux-gnu]
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick::HTTPServer#start: pid=1 port=3000
```

最后，我们需要创建一个数据库。打开另一个终端，运行如下命令：

```
$ fig run web rake db:create
```

当上述所有步骤都已经成功运行后，我们就可以通过访问宿主主机的3000端口来验证应用是否可以正常使用，如图15-1所示。

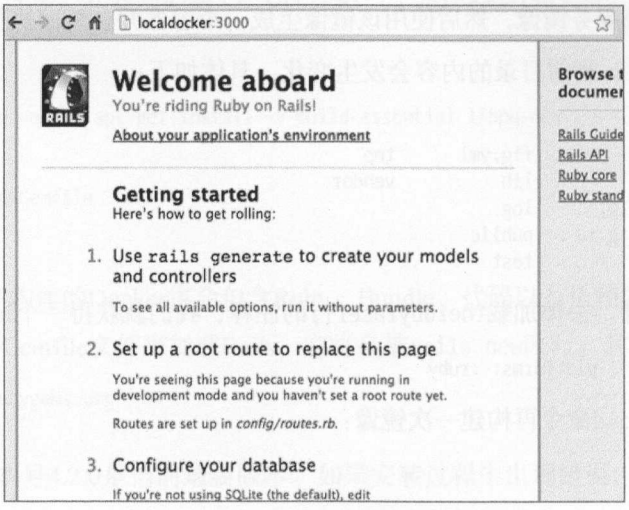


图15-1　Rails项目效果图

15.4　Django 开发环境配置

在这一节中，我们将使用Fig来配置一套运行Django/PostgreSQL的应用程序。

首先，新建一个项目目录，并在目录里创建3个文件。第一个文件是Dockerfile，用来描述安装在Docker容器里的软件依赖关系，其内容如下：

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
```

```
RUN pip install -r requirements.txt
ADD . /code/
```

该Dockerfile定义了一个镜像，它基于python:2.7镜像，如果本地没有该Python镜像，Docker会从Docker Hub中直接拉取。该文件还会安装requirements.txt文件中指定的依赖包。

第二个文件是requirements.txt，它定义了Python的依赖关系，其内容为：

```
Django
psycopg2
```

比较非常简单，只有两行，每一行都是Python的依赖包。最后，需要一个文件将这些配置都连接起来，它就是fig.yml文件，具体内容为：

```
db:
    image: postgres
web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
        - .:/code

    ports:
        - "8000:8000"

    links:
        - db
```

至此，我们就可以使用fig run命令来创建Django项目了：

```
$ fig run web django-admin.py startproject figexample .
```

执行该命令，Fig先会使用Dockerfile构建一个镜像，然后用该镜像创建Web容器，并在容器内部执行django-admin.py startproject figexample命令。运行完之后，就可以在当前目录下看到创建的新项目文件：

```
$ ls
Dockerfile      fig.yml         figexample      manage.py       requirements.txt
```

接下来，需要做的是修改Web到数据库的连接。修改figexample/settings.py 的DATABASES = ... 部分的内容为：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

此时会将默认的数据库连接改为连接到postgres数据库容器。

一切都配置好了，就可以在命令行中输入fig up来启动应用了：

```
$fig up
Recreating myapp_db_1...
Recreating myapp_web_1...
Attaching to myapp_db_1, myapp_web_1
myapp_db_1 |
myapp_db_1 | PostgreSQL stand-alone backend 9.1.11
myapp_db_1 | 2014-01-27 12:17:03 UTC LOG:  database system is ready to accept connections
myapp_db_1 | 2014-01-27 12:17:03 UTC LOG:  autovacuum launcher started
myapp_web_1 | Validating models...
myapp_web_1 |
myapp_web_1 | 0 errors found
myapp_web_1 | January 27, 2014 - 12:12:40
myapp_web_1 | Django version 1.6.1, using settings 'figexample.settings'
myapp_web_1 | Starting development server at http://0.0.0.0:8000/
myapp_web_1 | Quit the server with CONTROL-C.
```

这时开启浏览器，输入localhost:8000，就可以访问这个Django应用了。

15.5　WordPress 开发环境配置

Fig一样可以用在PHP应用的开发上，本节将介绍如何使用Fig来配置WordPress开发环境。

首先，下载WordPress源码，并将其解压到当前目录：

```
$ curl https://wordpress.org/latest.tar.gz | tar -xvzf -
```

目前，WordPress的最新版本为4.1。以上这条命令会创建名为wordpress的目录。进入此目录，创建镜像文件 Dockerfile，其内容如下：

```
FROM orchardup/php5
ADD . /code
```

这个Dockerfile直接基于orchardup/php5并把当前目录的code子目录加入到容器内部，用于加载WordPress的代码。

接着创建fig.yml，它将定义web服务和db服务，其中db服务基于mysql镜像。该文件的具体内容为：

```
web:
    build: .
    command: php -S 0.0.0.0:8000 -t /code
    ports:
        - "8000:8000"

    links:
        - db
```

```
    volumes:
      - .:/code

  db:
    image: mysql
    environment:
      MYSQL_DATABASE: wordpress
      MYSQL_ROOT_PASSWORD: wordpress
```

要让WordPress应用跑起来，还需要新建两个文件。一个为wp-config.php，它是WordPress的配置文件，这里我们仅需要修改其数据库连接部分，具体为：

```
<?php
    define('DB_NAME', 'wordpress');
    define('DB_USER', 'root');
    define('DB_PASSWORD', '');
    define('DB_HOST', "db:3306");
    define('DB_CHARSET', 'utf8');
    define('DB_COLLATE', '');
    define('AUTH_KEY',         'put your unique phrase here');
    define('SECURE_AUTH_KEY',  'put your unique phrase here');
    define('LOGGED_IN_KEY',    'put your unique phrase here');
    define('NONCE_KEY',        'put your unique phrase here');
    define('AUTH_SALT',        'put your unique phrase here');
    define('SECURE_AUTH_SALT', 'put your unique phrase here');
    define('LOGGED_IN_SALT',   'put your unique phrase here');
    define('NONCE_SALT',       'put your unique phrase here');
    $table_prefix  = 'wp_';
    define('WPLANG', '');
    define('WP_DEBUG', false);
    if ( !defined('ABSPATH') )
        define('ABSPATH', dirname(__FILE__) . '/');
    require_once(ABSPATH . 'wp-settings.php');
```

我们定义了数据库相关的配置，并把数据库连接为db容器。另一个文件是router.php，它定义PHP内置的Web服务器如何运行WordPress，其内容为：

```
<?php
    $root = $_SERVER['DOCUMENT_ROOT'];
    chdir($root);
    $path = '/'.ltrim(parse_url($_SERVER['REQUEST_URI'])['path'],'/');
    set_include_path(get_include_path().':'.__DIR__);
    if(file_exists($root.$path))
    {
        if(is_dir($root.$path) && substr($path,strlen($path) - 1, 1) !== '/')
            $path = rtrim($path,'/').'/index.php';
        if(strpos($path,'.php') === false) return false;
        else {
        chdir(dirname($root.$path));
        require_once $root.$path;
    }
    }else include_once 'index.php';
```

当所有文件都配置好后，就可以使用fig up来启动WordPress了。然后使用浏览器访问localhost:8000，就可以浏览到WordPress的安装首页。

15.6　Flocker：跨主机的 Fig 应用

Fig是一个在单台机器上管理多个容器的利器。然而有些应用，例如ELK（ElasticSearch-Logstash-Kibana）应用，需要运行在集群的多台机器上，这时我们要寻找一种在多台机器上的解决方案。Flocker，一个协助Fig在多机器上部署应用的工具，能够有效地管理多机器的多容器和数据卷。Flocker支持Fig的yml文件格式和语法，所以你可以使用Fig配置好应用，然后使用Flocker将其部署到集群的多节点上。Flocker的亮点不仅在于能够多节点部署，更为值得一提的是它的数据管理功能。当容器内部有数据库运行时，因为数据的重要性，我们需要对容器格外小心，需要经常做迁移、备份等工作，而Flocker有一套很好的工具来完成这些事情。

首先，我们需要安装Flocker组件，它包含两部分，一部分是客户端flocker-cli，它通过SSH方式控制集群的所有节点；另一部分是flocker-node，flocker集群里面的每个节点都需要安装，它包含flocker-changestate、 flocker-reportstate和flocker-volume，这些组件由flocker-deploy命令来操作，用于完成容器和容器数据卷的迁移等工作。

1.安装 flocker-cli

在安装flocker-cli之前，我们需要安装好Python等依赖包。在Red Hat和Fedora系列中，其操作如下：

```
$ sudo yum install @buildsys-build python python-devel python-virtualenv
```

在Ubuntu或Debian系统下，则是：

```
$ sudo apt-get install gcc python2.7 python-virtualenv python2.7-dev
```

然后新建一个sh脚本，其内容为：

```
#!/bin/sh
# Create a virtualenv, an isolated Python environment, in a new directory called
# "flocker-tutorial":
virtualenv --python=/usr/bin/python2.7 flocker-tutorial
# Upgrade the pip Python package manager to its latest version inside the
# virtualenv. Some older versions of pip have issues installing Python wheel
# packages.
flocker-tutorial/bin/pip install --upgrade pip
# Install flocker-cli and dependencies inside the virtualenv:
echo "Installing Flocker and dependencies, this may take a few minutes with no output to the terminal..."
flocker-tutorial/bin/pip install --quiet
https://storage.googleapis.com/archive.clusterhq.com/downloads/flocker/Flocker-0.3.2-py2-none-any.
whl
echo "Done!"
```

将其保存为linux-install.sh，然后执行如下命令：

```
$ sh linux-install.sh
...
```

程序运行完之后，flocker-cli的二进制文件会在当前目录的flocker-tutorial/bin/目录下，我们可以将它们复制到/usr/bin下，然后验证安装是否正确：

```
$ flocker-deploy --version
```

2. 安装flocker-node

这里我们以Red Hat和Fedora系列为例。首先需要安装好其依赖包，主要有zfs和kernel-devel。安装kernel-devel包时，需要安装和当前系统对应的版本。下面的所有操作都是在需要安装flocker-node的机器上执行的：

```
UNAME_R=$(uname -r)
PV=${UNAME_R%.*}
KV=${PV%%-*}
SV=${PV##*-}
ARCH=$(uname -m)
yum install -y
https://kojipkgs.fedoraproject.org/packages/kernel/${KV}/${SV}/${ARCH}/kernel-devel-${UNAME_R}.rpm
yum install -y https://s3.amazonaws.com/archive.zfsonlinux.org/fedora/zfs-release$(rpm -E
    %dist).noarch.rpm
yum install -y http://archive.clusterhq.com/fedora/clusterhq-release$(rpm -E %dist).noarch.rpm
    yum install -y flocker-node
```

在每一个节点上都执行上述操作，flocker-node就安装完毕了。接着，我们需要启动它并将它加入开机启动：

```
systemctl start docker
systemctl enable docker
```

为了能够让节点直接相互转发数据，我们需要配置防火墙，允许节点间的数据通信，具体操作为：

```
firewall-cmd --permanent --direct --add-rule ipv4 filter FORWARD 0 -j ACCEPT
firewall-cmd --direct --add-rule ipv4 filter FORWARD 0 -j ACCEPT
```

此外，还得创建名为flocker的ZFS池，具体操作为：

```
mkdir /opt/flocker
truncate --size 10G /opt/flocker/pool-vdev
zpool create flocker /opt/flocker/pool-vdev
```

此外，必须保证flocker-cli的组件flocker-deploy和每一个节点都能够建立SSH连接。不仅如此，每个节点直接也要建立SSH连接，并且保证每个节点的22号端口是允许访问的。flocker-cli客户端可以以root身份登录到所有节点上，并将用户的SSH公钥加入到各个节点root用户的~/.ssh/authorized_keys中。假设A主机上的root用户想通过无密码登录B主机，那么需要在A主机上生成

该用户的一对密钥。该密钥包含私钥和公钥，私钥由A主机保管好，公钥则需要上传到主机B上，并将其导入到root用户的~/.ssh/authorized_keys中。在A主机上的操作如下：

```
$ ssh-keygen -t rsa
$scp ./id_rsa.pub  root@104.131.93.66:/home/root/
```

将上述IP地址换成B的IP地址，然后在B主机上导入该公钥：

```
~# cat id_rsa.pub >> ~/.ssh/authorized_keys
```

接着在A主机上运行ssh来登录B：

```
$ ssh root@104.131.93.66
```

第二次登录时，就不再需要密码了。

Flocker安装好之后，接下来通过Flocker来部署一个ELK应用。

3. 使用fig.yml来构建ELK应用

ELK应用包含3个组件——ElasticSearch、Logstash和Kibana，我们将为每个组件构建一个容器并将它们连接在ElasticSearch容器中。此外，我们还会挂载数据卷。这3个组件的连接如下。

- Logstash接受日志消息并将它们投递给ElasticSearch。
- ElasticSearch将日志保存在数据库中。
- Kibana将连接到ElasticSearch，检索日志信息并将其展现到Web页面上。

在fig.yml文件中，我们仅需要定义好应用组件，而不用告诉每个组件在哪台机器上运行。默认情况下，Fig会将应用部署到本机上。fig.yml文件的具体内容为：

```
elasticsearch:
    image: clusterhq/elasticsearch
    ports:
        - "9200:9200"
    volumes:
        - /var/lib/elasticsearch

logstash:
    image: clusterhq/logstash
    ports:
        - "5000:5000"

    links:
        - elasticsearch:es

kibana:
    image: clusterhq/kibana
    ports:
        - "80:8080"
```

4. 使用Fig在单机上部署开发环境

配置好YAML文件后，就可以通过fig up命令将应用部署到本机上：

```
$ fig up
```

Fig会去下载相应的镜像并启动基于镜像的容器。如果需要将容器放到后台运行，那么可以在fig up命令后加上-d参数。

5. 使用Flocker将ELK部署到多节点上

接下来，需要将ELK部署到不同的服务器上，这里我们使用两台虚拟机，系统为Fedora 20。我们需要一个部署配置文件deployment.yml，在这个文件中我们指定哪个容器将会被部署到哪个IP上：

```
"version": 1
"nodes":
    "104.131.93.66": ["logstash", "kibana"]
    "104.131.98.216": ["elasticsearch"]
```

保存好文件之后，我们使用flocker-deploy工具执行如下操作：

```
$ flocker-deploy deployment.yml fig.yml
```

这样，Logstash和Kibana被部署到同一台机器上，ElasticSearch被部署到另一台机器上。

6. 使用Flocker进行数据迁移

随着应用执行时间的推移，记录的日志也越来越多。ElasticSearch的磁盘已经装不下那么多数据，这时需要将它迁移到另外一个存储更大的新节点上。Flocker把这个迁移过程做得非常简单，你只需要重新修改下deployment.yml文件，然后再重新运行flocker-deploy即可。将deployment.yml文件的内容修改为：

```
"version": 1
"nodes":
    "104.131.93.66": ["logstash", "kibana"]
    "104.131.98.216": []
    "104.131.7.13": ["elasticsearch"]
```

然后重新运行flocker-deploy命令即可。我们可以分别在原来的ElasticSearch机器和新的ElasticSearch机器上查询其日志，具体操作如下：

```
marcus@client:~/elk$ flocker-deploy deployment.yml fig.yml
marcus@client:~/elk$ ssh root@104.131.98.216<script cf-hash="f9e31" type="text/javascript">
/* <![CDATA[ */!function(){try{var t="currentScript"in
document?document.currentScript:function(){for(var
t=document.getElementsByTagName("script"),e=t.length;e--;)if(t[e].getAttribute("cf-hash"))return
t[e]}();if(t&&t.previousSibling){var
e,r,n,i,c=t.previousSibling,a=c.getAttribute("data-cfemail");if(a){for(e="",r=parseInt(a.substr(0,
2),16),n=2;a.length-n;n+=2)i=parseInt(a.substr(n,2),16)^r,e+=String.fromCharCode(i);e=document.cre
ateTextNode(e),c.parentNode.replaceChild(e,c)}}}catch(u){}}();/* ]]> */</script> docker ps
```

```
CONTAINER ID    IMAGE                                          COMMAND                  CREATED
STATUS          PORTS                                          NAMES


marcus@client:~/elk$ ssh root@104.131.7.13<script cf-hash="f9e31" type="text/javascript">
/* <![CDATA[ */!function(){try{var t="currentScript"in
document?document.currentScript:function(){for(var
t=document.getElementsByTagName("script"),e=t.length;e--;)if(t[e].getAttribute("cf-hash"))return
t[e]}();if(t&&t.previousSibling){var
e,r,n,i,c=t.previousSibling,a=c.getAttribute("data-cfemail");if(a){for(e="",r=parseInt(a.substr(0,
2),16),n=2;a.length-n;n+=2)i=parseInt(a.substr(n,2),16)^r,e+=String.fromCharCode(i);e=document.cre
ateTextNode(e),c.parentNode.replaceChild(e,c)}}}catch(u){}}();/* ]]> */</script> docker ps


CONTAINER ID    IMAGE                                          COMMAND                  CREATED
STATUS          PORTS                                          NAMES


54e26b1e4043    clusterhq/elasticsearch:latest                 "/bin/sh -c 'source    23 seconds ago
Up 22 seconds   9300/tcp, 0.0.0.0:9200-&gt;9200/tcp            flocker--elasticsearch


marcus@client:~/elk$ curl -XGET --silent 'http://104.131.98.216:9200/_search?q=firstname:Joe' |
    underscore extract hits.hits


[
    {
        "_index": "logstash-2014.10.16",
        "_type": "logs",
        "_id": "KWqBIoOAQEm4S6Q9oFOMyQ",
        "_score": 0.30685282,
        "_source": {
            "firstname": "Joe",
            "lastname": "Bloggs",
            "@version": "1",
            "@timestamp": "2014-10-16T08:04:57.568Z",
            "host": "104.131.91.191:54904"
        }
    }
]
```

可以看到，旧的数据已经成功迁移到了新的机器上。这里我们简单说一下flocker-deploy在迁移容器和数据卷中发生了什么。首先，Flocker会给容器的数据卷生成一个快照，然后把该快照发送给新的主机；旧的容器会被停止，在快照之后的部分数据会被发送给新的主机；然后在新的主机上启动容器，数据库连接等设置自动被重新设定，指向新的数据库容器。因为Flocker采用了分步骤的迁移数据：第一步用于生成和迁移快照，数据量大，复制时间长，但这段时间容器并没有停止工作；第二步复制剩下的部分数据，量非常小，所以容器宕机的时间很短。

第 16 章 Kubernetes 16

在第15章中，我们介绍了Fig和Flocker，它们都是为了更好地管理Docker的容器。在本章中，我们将会介绍另外一款大规模容器集群的管理工具——Kubernetes，它是由Google公司开发的开源软件，其核心采用Go语言开发，代码在GitHub上维护。目前，Docker、微软、IBM以及RedHat等公司已经加入到Kubernetes社区。本章将介绍如下内容。

- Kubernetes简介。
- Kubernetes核心概念，主要包含节点、Pod、服务、控制器、卷和标签等。
- Kubernetes的架构和组件。Kubernetes是一种主从架构的集群管理工具，主控节点包含apiserver、调度器和控制器，从属节点包含kubelet和服务代理等组件。
- Kubernetes实战。

16.1 Kubernetes 简介

Kubernetes是Google公司开源的大规模容器集群管理系统。利用Kubernetes，能方便地管理跨机器运行的容器化应用。它为容器化应用提供资源调度、部署、服务发现、扩展机制等功能，具体如下：

- 使用Docker对应用进行打包、实例化和运行；
- 以集群的方式运行和管理跨主机的容器；
- 解决跨主机容器的通信问题；
- 提供自我修复功能，保证系统运行的健壮性。

Kubernetes目前处于快速迭代开发之中，几乎每周都会推出新的版本。作为一个容器管理框架，Kubernetes可以被部署在物理集群和各类云环境中，例如GCE、vSphere、CoreOS、OpenShift、Azure等。

接下来，我们将会从Kubernetes的核心概念、设计框架等入手，让读者更好地理解它。等有了相关基础之后，我们将会在CentOS 7上实验Kubernetes。

16.2　核心概念

Kubernetes的核心概念包含节点（Node）、Pod、服务（Service）、备份控制器（Replication Controller）、卷（Volume）和标签（Label）。鉴于备份控制器和卷比较简要，这里不再详细介绍，后面介绍架构时一起说明。

16.2.1　节点

节点是Kubernetes系统中的一台工作机器，常被称为Minion，即从属主机。它可以是物理机，也可以是虚拟机。每一个节点都包含了Pod运行所需的必要服务，例如Docker、kubelet和网络代理（proxy）。节点受Kubernetes系统中的主节点控制。和Pod、服务不一样，节点本身并不属于Kubernetes的概念，它是云平台中的虚拟机或者实体机。所以，当一个节点加入到Kubernetes系统中时，它将会创建一个数据结构来记录该节点的信息。另外，不是所有节点都能够加入到Kubernetes系统中的，只有那些通过验证的节点才能够成为Kubernetes节点。

目前，节点的管理有两种方式：节点管理器（Node Controller）和通过命令手动管理。

- **节点管理器。**它是Kubernetes主控节点上管理集群节点的组件，主要包含两个功能：集群节点的同步和单个节点生命周期的管理。当有节点加入到Kubernetes中时，节点管理器将会创建节点信息；当有节点需要从Kubernetes中删除时，节点管理器则会删除该节点的节点信息。需要注意的是，节点管理器并不会真正创建节点本身，而仅仅创建节点的元数据，用于跟踪节点的状态。所以，节点上的服务需要用户自己安装。单个节点的生命周期的管理目前尚在开发之中。
- **手动管理节点。**Kubernetes的管理员可以通过kubectl命令来管理节点。和节点管理器一样，使用kubectl命令创建和删除节点时，也只是删除节点的配置信息。

16.2.2　Pod

在Kubernetes中，Pod是最小的可创建、调度和管理的部署单元。它是容器化环境中的“逻辑主机”，可以包含一个或多个有关联的容器，并且容器之间可以共享数据卷。例如，一个Web站点应用由前端、后端和数据库组成，这三个组件运行在各自的容器中，我们可以创建包含这三个容器的Pod。

可以看出，容器存在于Pod之中，而Pod又存在于节点之中。那么，为什么需要抽象出Pod这个概念呢？下面从资源共享和通信、管理这两个方面介绍一下。

- **资源共享和通信。**同一Pod中的容器拥有相同的网络命名空间、IP地址和端口区间，它们之间可以直接用localhost来发现和通信。在无层次的共享网络中，每个Pod都有一个IP地址，用于跟其他物理主机和容器进行通信，Pod的名字也被用作主机名。此外，同一Pod的容器可以共享数据卷。在将来，Pod内的容器还可以共享IPC命名空间、CPU和内存等。

- **管理**。从管理的角度来看，Pod比容器站在更高的层面，它简化了应用的部署和管理。Pod可以自动处理主机托管、资源共享、协调复制和依赖管理等问题。

虽然可以将Pod用在垂直依赖的应用栈中，但它更适合用在多个应用的横向协作部署中，为多个容器提供集中的辅助功能。具体的使用用例有：

- 内容管理系统、文件和数据的装载和本地缓存管理等；
- 日志和检出点备份、压缩、轮换和快照；
- 数据变更监控、日志末端数据读取、日志和监控适配器和事件打印；
- 代理、桥接和适配器；
- 控制器、管理器、配置编辑和更新。

为什么不在一个容器中直接运行多个应用而采用在一个Pod中运行多个容器呢？主要原因如下所示。

- **透明性**。底层系统可以获取到Pod内的容器，这样底层系统就可以为容器提供诸如进程管理和资源监控等服务，这会给用户带来不少便利。
- **解耦软件依赖**。分为多个容器后，每个容器都可以单独存在，并且当其中某个应用需要升级时，只影响到一个容器。后续Pod将会支持单个容器的在线升级。
- **易用性**。用户不需要使用自己的进程管理程序，直接用Docker管理容器即可。此外，用户也不用再担心信号量和退出码的传递等问题。
- **高效性**。因为底层系统提供了更多的管理，这使得容器更加轻便。

16.2.3 服务

Kubernetes的服务是一系列Pod以及这些Pod的访问策略的抽象。

Kubernetes中的Pod是具有时效性的，它会随着时间而变化。虽然每个Pod都有一个单独的IP地址，但是该IP地址却不是静态不变的。例如，当系统触发RepliController对Pod进行备份时，Pod的地址很可能会发生改变。这将会导致一个问题。我们试想，在Kubernetes系统中，有一群Pod作为后端给前端提供服务，如果后端的IP地址是变动的，那么前端又如何去发现和使用后端的服务呢？

Kubernetes的服务也叫作微服务（micro-service），它用于定义一系列Pod的逻辑关系以及它们的访问规则。服务的目标是为了隔绝前端和后端的耦合性，让前端透明地使用该项服务，而不需要知道该项服务具体由哪些后台机器提供。Kubernetes会为一个服务分配一对，该IP和端口并不是真实的地址和端口，而是一个虚拟IP，当前端通过该对访问服务时，服务代理将会将请求重定向到合适的后端机器。

1. 定义服务

Kubernetes中的服务是一个REST（Representational State Transfer，表述性状态转移）对象。下面演示了一个服务的定义：

```
{
    "id": "myapp",
    "selector": {
        "app": "MyApp"
    },
    "containerPort": 9376,
    "protocol": "TCP",
    "port": 8765
}
```

上述示例定义了一个id为myapp的服务，它通过选择器选择那些带有app=MyApp标签的Pod作为服务的提供者。所有被选择的Pod都将9376端口暴露，用于监听该项服务的请求。前端客户可以通过$MYAPP_SERVICE_PORT和$MYAPP_SERVICE_HOST来访问该项服务。

2. 工作原理

在Kubernetes中，每个节点都运行着一个服务代理（service proxy），它监控来自Kubernetes主控节点的消息，主控节点会向其传递诸如添加和删除服务以及服务的端点列表等信息。服务代理维护着一个映射表，表中每一项是服务和该服务的提供者列表的映射关系，服务代理还会为每一个服务在本地开放一个端口，当节点需要使用某项服务时，将请求发送至该端口，然后由服务代理通过某种策略（例如轮换策略）安排服务的具体提供节点。

当一个Pod加入到Kubernetes集群中时，主控节点会在它上面为每一个已经存在的服务分配一系列环境变量。变量的命名形如SVCNAME_SERVICE_HOST，其中SVCNAME是服务名称的大写。例如，服务redis-master在端口6379上提供TCP服务，其虚拟IP地址为10.0.0.11，那么该项服务对应的环境变量就有：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

这些环境变量主要说明服务的地址、端口和协议。需要注意的是，既然一个Pod加入到Kubernetes集群时，会被设置这些跟服务相关的环境变量，这意味着一个Pod只能发现在它加入之前就已经存在的服务。当然，如果服务支持了DNS，该条限制将会失效。

如图16-1所示，在前端Pod加入到该Kubernetes集群时，主控节点的apiserver会将MyApp服务的服务信息推送给该Pod，这些信息包含了服务及其对应的端点（Endpoint）列表。前端Pod需要使用后端Pod提供的MyApp服务，它并不直接联系后端Pod，而是将请求发给本地的服务代理，服务代理会将该服务请求分配给这三个后端Pod中的一个。服务可以动态地增加和删除提供服务的Pod，而前端Pod感知不到这些细节的变化，除非是正在为它提供服务的Pod状态发生了改变。

在Kubernetes服务的工作原理背后，还有两个细节需要注意：冲突避免和Portal。

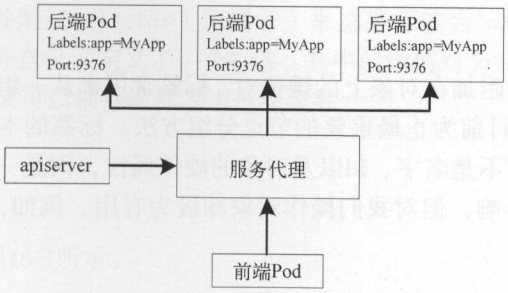


图16-1 前端通过服务代理来访问后端提供的服务

3. 冲突避免

Kubernetes的一个设计理念就是不能随便让用户去承受服务的失败，特别是用户根本就没有犯错的情况下。这里我们考虑一下服务端口冲突的问题：假如第一项服务选取了80端口作为服务端口，那么其他服务就不能再选该端口作为服务端口了。为了避免这种冲突问题，Kubernetes不仅选择了为每一个服务分配端口，同时也配置了一个IP。这样每一个服务都拥有自己的IP和端口对应，从而避免了服务端口冲突。

4. Portal

Portal就是前面提到的服务的二元组。这里的IP是虚拟IP，它并不是一台特定机器的真实IP。用户访问某个服务，就是访问某个服务的Portal。当请求投递到该Portal之后，该请求会根据规则重定向到某个特定的服务提供者Pod。下面我们举例说明该问题。如图16-2所示，每一个节点都会有一个服务代理，当节点加入到Kubernetes集群中时，主控节点的apiserver程序会将服务的配置信息投递给服务代理，服务代理保存服务配置信息，并根据Portal信息设置iptables的网络规则，这里我们假设Portal为10.0.0.1:1234。前端通过链接Portal来访问服务，iptables监听到该请求，将该请求重定向到配置好的代理端口上，服务代理从该端口接过服务请求，然后根据策略选取一个后端Pod，最后由选中的后端Pod来给前端Pod提供服务。

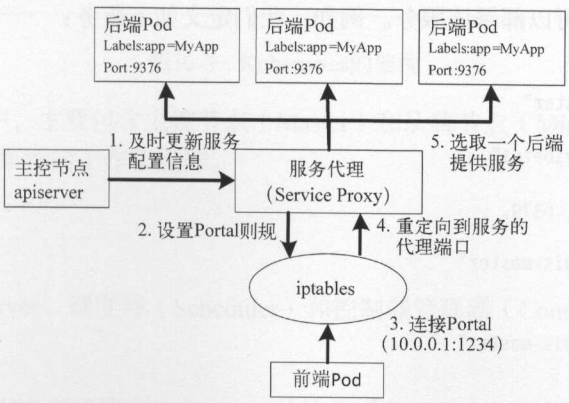


图16-2 服务代理响应服务的过程

16.2.4　标签

标签（Label）是一组附加在对象上的键值对。标签常用来从一组对象中选取符合条件的对象，这也是Kubernetes中目前为止最重要的节点分组方法。标签的本质是附属在对象上的非系统属性类的元数据，即它不是名字、Id以及对象的硬件属性，而是一些附加的键值对，这些键值对对对象本身没什么影响，但对我们操作对象却极为有用。例如，下面是一个Pod节点的配置文件：

```
{
    "id": "redis-master",
    "kind": "Pod",
    "apiVersion": "v1beta1",
    "desiredState": {
        "manifest": {
            "version": "v1beta1",
            "id": "redis-master",
            "containers": [{
                "name": "master",
                "image": "dockerfile/redis",
                "cpu": 100,
                "ports": [{
                    "containerPort": 6379,
                    "hostPort": 6379
                }]
            }]
        }
    },
    "labels": {
        "name": "redis-master"
    }
}
```

可以看到，除了id、kind等这些系统属性外，还有labels属性，它是这个Pod的标签，它的键值对为"name":"redis-master"。有了这个标签，我们就可以在部署服务时，通过标签来指定只有包含该标签的Pod才可以部署该服务。例如，我们定义如下服务：

```
{
    "id": "redis-master",
    "kind": "Service",
    "apiVersion": "v1beta1",
    "port": 6379,
    "containerPort": 6379,
    "selector": {
        "name": "redis-master"
    },
    "labels": {
        "name": "redis-master"
    }
}
```

这个redis-master服务通过selector（选择子）来选择标签为"name":"redis-master"的Pod来部署服务。当然，该服务自己又定义了一个标签，其中的键值对为"name":"redis-master"，这样需要使用该服务的应用又可以根据该标签来过滤服务。

16.3 架构和组件

Kubernetes的架构如图16-3所示。

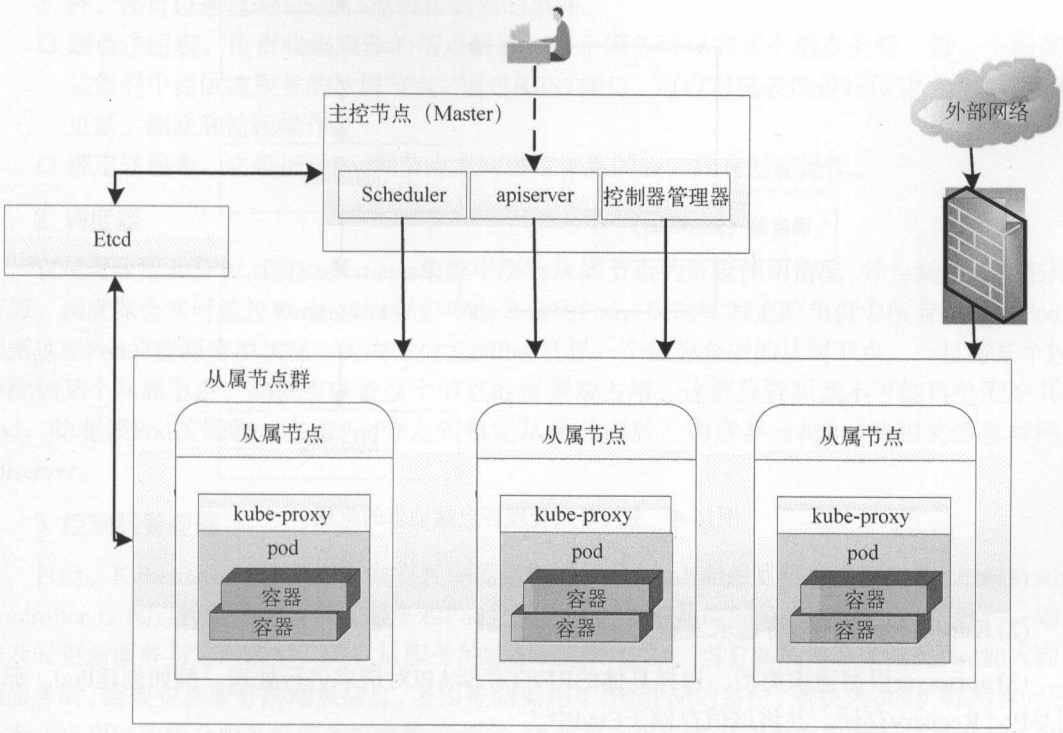


图16-3 Kubernetes的架构

在Kubernetes集群中，主要包含主控节点（Master）和从属节点（Minion），前者负责整个集群的管理工作，后者是集群的工作节点群。

16.3.1 主控节点

主控节点包含apiserver、调度器（Scheduler）和控制器管理器（Controller Manager）。

1. apiserver

在apiserver中，我们定义了诸多Kubernetes的核心对象以及它们的操作。这些核心对象包含

Pod注册表（Pod Registry）、控制器注册表（Controller Registry）、服务注册表（Service Registry）、端点注册表（Endpoint Registry）、从属注册表（Minion Registry）、绑定注册表（Binding Registry）。在分别说明这些注册表之前，我们先来看看主控节点是如何处理客户端请求的，具体如图16-4所示。

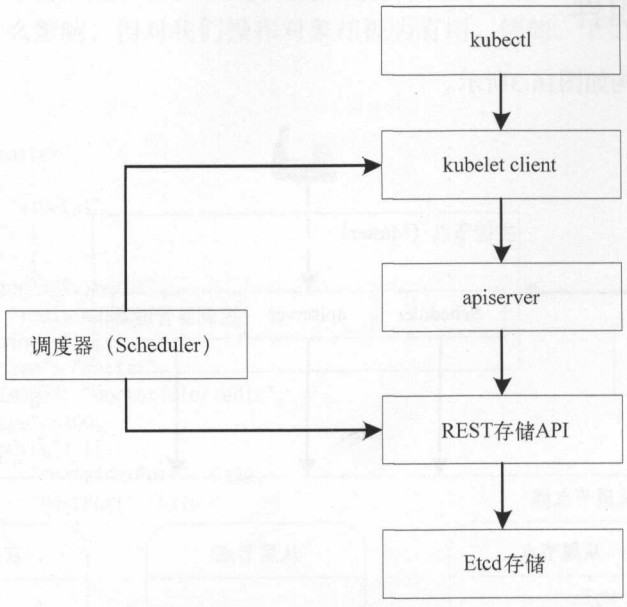


图16-4　主控节点处理客户端请求的流程

(1) kubectl将用户命令发送给Kubernetes客户端。

(2) Kubernetes客户端将请求发送给apiserver。

(3) apiserver根据请求类型，选择具体的REST存储API对请求进行处理。例如创建Pod，那么则是Pod Registry存储，并将该值存储于Etcd中。

(4) 在apiserver响应请求之后，调度器会去Kubernetes客户端收集从属节点的节点信息以及Pod信息。

(5) 根据所收集的信息，调度器将新建的Pod分发到合适的从属节点上。

在apiserver的存储库中，存储着Kubernetes的各种核心对象信息，它们是一个个注册表。下面我们简要说明最重要的几个注册表。

- **从属注册表**。负责跟踪集群中的从属节点信息。Kubernetes将节点注册表的信息封装成RESTful形式并提供API，通过这些API可以创建和删除从属节点，目前不支持从属节点的修改。Scheduler根据从属的节点信息决定是否将新的Pod分配到该节点。
- **Pod注册表**。记录集群中的Pod信息以及Pod和从属节点的映射关系。通过REST接口，可

以对Pod进行创建（Create）、获取（Get）、列出（List）、更新（Update）和删除（Delete）等操作。此外，可以通过watch接口监听Pod的事件，例如创建、删除等。

- **服务注册表。**负责跟踪集群中所有服务的信息。通过REST接口，可以对服务进行创建、获取、列出、更新和删除等操作。此外，也可以通过watch接口设置事件监听，监听服务的变更事件。
- **控制器注册表。**负责跟踪集群中所有的控制器，例如备份控制器（Replication Controller）的信息。通过REST接口，可以对控制器进行创建、获取、列出、更新和删除等操作。此外，还可以通过watch接口监听控制器的事件。
- **端点注册表。**负责收集服务的端点信息。一个服务可以和多个端点关联，每一个端点就是集群中提供该服务的从属节点。通过REST接口，可以对该表项进行创建、获取、列出、更新、删除和监视操作。
- **绑定注册表。**它是记录Pod和节点之间绑定关系的表，只有创建操作。

2. 调度器

它负责收集和分析当前Kubernetes集群中所有从属节点的资源使用情况，并为新建的Pod分配资源。调度器会实时监控Kubernetes集群中未分发的Pod，同时实时追踪集群中所有运行的Pod，根据这些Pod的资源使用状况，为尚未分发的Pod选择一个最为合适的从属节点。一旦将某个Pod分配到某个从属节点，那就意味着这个节点的资源被占用，这部分资源就不可能再分配给其他Pod，除非该Pod被回收。在将Pod分发到指定从属节点后，调度器会把Pod的相关信息写回到apiserver。

3. 控制器管理器

目前，Kubernetes的控制器有端点控制器（Endpoint Controller）和备份控制器（Replication Controller）。端点控制器主要保证服务和Pod之间的映射关系是最新的。例如当Pod失效时，则应该及时更新服务与它的映射，将它从服务的映射列表中去除。当有新的符合条件的Pod加入到一项服务时，需要更新服务的端点信息。备份控制器用于控制Pod的备份，解决Pod的扩容缩容问题。分布式应用出于提升服务性能和容错性等考虑，需要复制多份资源并根据负载而动态增加或者减少。当某项服务的某个Pod因为异常而宕机时，备份控制器在检测到该事件发生后，会立即新建一个该服务的Pod，以保证服务质量。

备份控制器的主要用法如下所示。

- **调度。**备份控制器会保证指定Pod的指定副本数量在运行，当节点异常退出时，立即新建Pod副本进行替换。
- **扩容缩容。**根据需要动态增加或减少Pod数量。
- **逐步更新。**某项服务需要更新时，可以对一个个的Pod进行升级更新。
- **应用的多分支跟踪。**一个应用可以有多个分支，并且在集群中可以同时运行多个分支，而分支之间通过标签来识别。

16.3.2　从属节点

从属节点是Kubernetes集群中真正工作的节点。除了包含Pod外，还有用于管理和通信的基础设施，主要是kubelet组件和服务代理。

1. kubelet

kubelet主要负责管理Pod及容器，以及与apiserver通信。它接收来自主控节点的apiserver组件发来的命令和任务，并与Etcd、http等服务交互。kubelet包含Docker客户端、根目录、Pod Worker、Etcd客户端、Cadvisor客户端以及Health Checker组件。它的工作具体包括：

- 通过Pod Worker给Pod分配任务；
- 同步Pod的状态；
- 从Cadvisor获取容器、Pod、宿主主机信息；
- 管理Pod容器，包括运行一个指定的容器，创建网络容器，给容器绑定数据卷和端口，杀死和删除容器以及在容器中运行命令。

2. 服务代理

在16.2.3节中说明“服务”的概念时，我们提及服务代理，本节的代理即为服务代理。每生成一种服务，代理都会从Etcd中获取该服务的端点列表信息，然后根据配置设置iptables规则，对服务请求进行重定向。

16.3.3　组件交互流程

在介绍完核心概念、架构和组件之后，接下来说明这些组件是如何串联起来为我们服务的。这里我们介绍一下创建Pod、备份控制器和服务的流程。

1. 创建Pod的流程

图16-5展示的是创建Pod的时序图。首先用户发起一个创建Pod的请求，kubectl会将该请求发送给主控节点的apiserver组件；apiserver收到请求后，会向Etcd服务器请求添加一个Pod对象。调度器定时获取整个集群中从属节点和Pod的状态，收集其中的资源利用状况，然后决定将本次新建的Pod分配到哪个从属节点。当从属节点和Pod绑定后，调度器会将该绑定信息返回给apiserver，然后apiserver会将该绑定持久化到Etcd中。此后，从属节点上的kubelet会定时地向Etcd汇报该绑定的状态。kubelet会根据Pod配置信息创建并启动容器。

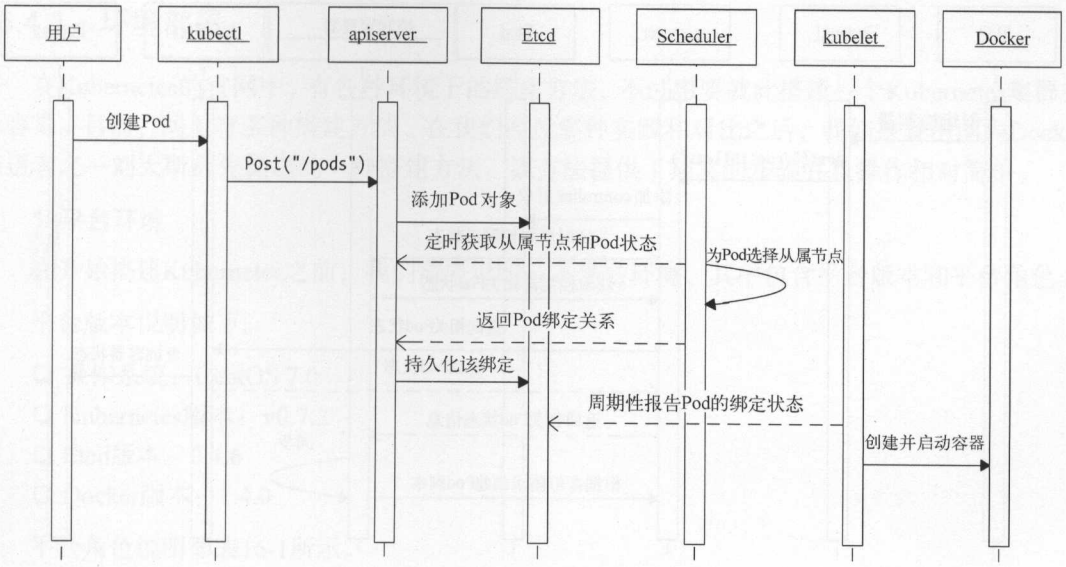


图16-5 创建Pod的时序图

2. 创建备份控制器的流程

图16-6是创建备份控制器的时序图。用户通过kubectl创建控制器，kubectl会将该请求投递到主控节点的apiserver组件上，apiserver会在Etcd上新建controller对象。控制器管理器定时从apiserver查询控制器的状态，获取控制器相关Pod的状态。当apiserver收到查询Pod状态的请求后，它需要向相关的从属节点发出Pod状态查询，从属节点上的kubelet组件收到该请求后，会进一步向Pod中的容器发起查询，然后将状态返回给apiserver，进一步返回给控制器管理器。控制器管理器同步好Pod状态信息后，如有需要，例如发现有些Pod已经异常退出了，那么它就需要创建Pod的副本，以保证集群中始终保持设定数量的Pod在运行。

3. 创建服务的流程

图16-7是创建服务的时序图。用户先创建一个新的应用服务，apiserver会通知Etcd建立相应的service对象。对于服务的定义，可以参见16.2.3节中的示例，其中有个标签选择器，通过它可以知道哪些Pod是会提供该服务的。控制器管理器获取该Pod列表后，将这些Pod都设定为该服务的端点，这些端点信息会保存在Etcd中。从属节点上的服务代理会定期从apiserver那里获取所有服务的信息，发现有新的服务并确认自己属于服务的提供者之后，接着在本地创建套接字用于监听该服务的请求，设置iptables的网络规则，用于服务的重定向。此外，代理会及时更新服务的端口列表，以供服务响应。

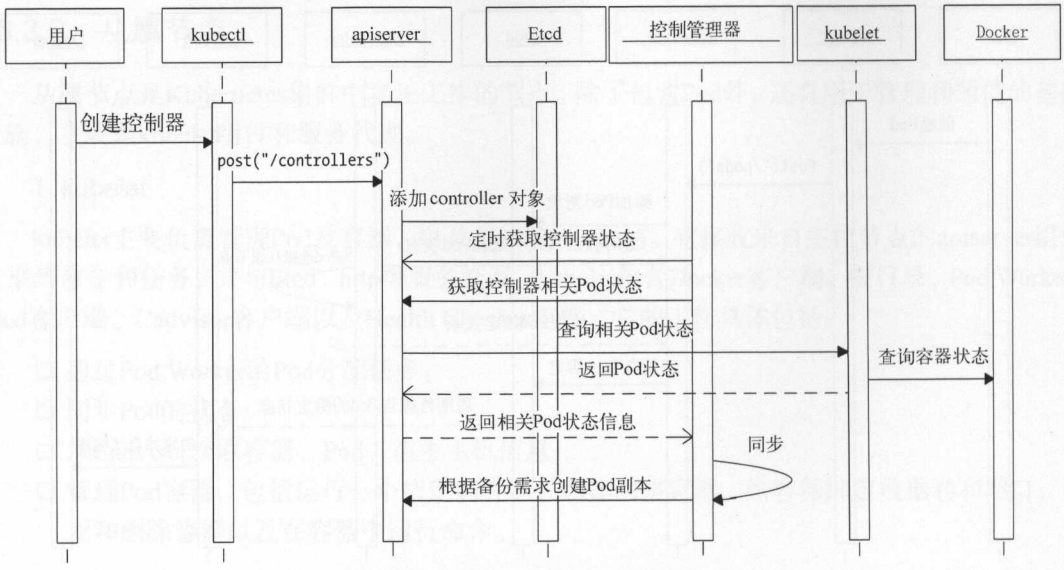


图16-6　创建备份控制器的流程

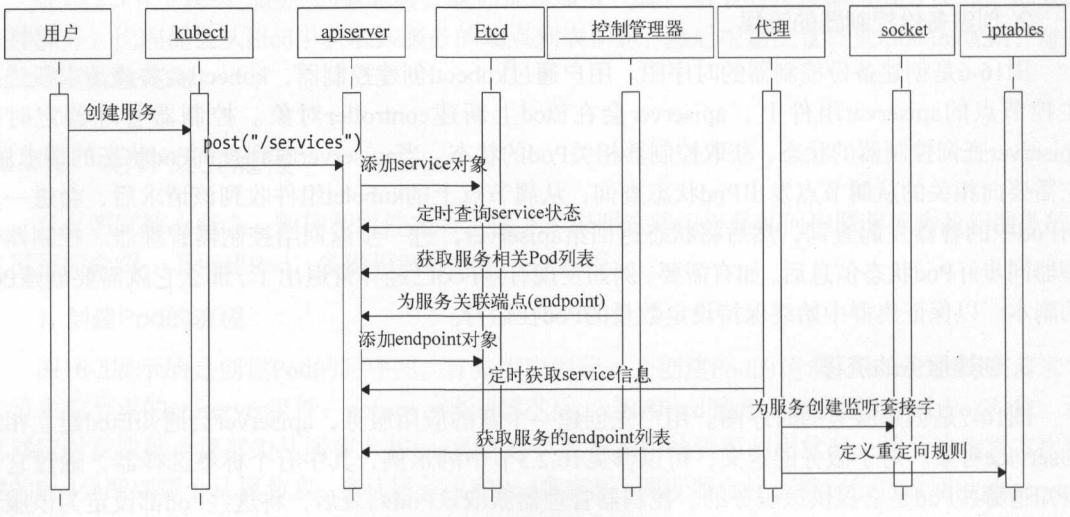


图16-7　创建服务的时序图

16.4　Kubernetes 实战

在这一节中，我们将通过实际操作来说明如何搭建一个Kubernetes集群，其中主要涉及环境部署和应用操作。

16.4.1 环境部署

在Kubernetes的官网中，有各种环境下的搭建方法，不过想要就此搭建一个Kubernetes集群并不容易。目前，网上有多种搭建方法，在我们经过多种实践和对比之后，我们推荐由国内Docker布道者之一刘天斯首先提出的一种搭建方法，该方法提供了翔实的步骤并且操作相对简单。

1. 平台环境

在开始搭建Kubernetes之前，我们需要说明一下平台环境，其中包含平台版本和平台角色。

平台版本说明如下。

- 操作系统：CentOS 7.0
- Kubernetes版本：v0.7.2
- Etcd版本：0.4.6
- Docker版本：1.4.0

平台角色说明如表16-1所示。

表16-1 平台角色

角　色	IP	组件说明
主控节点	192.168.1.83	Kubernetes
Etcd	192.168.1.84	Etcd
从属节点	192.168.1.85	Kubernetes+Docker
从属节点	192.168.1.86	Kubernetes+Docker

2. 环境安装

环境安装包含系统环境安装以及软件的安装。在下面的安装过程中，读者需要特别注意不同主机因为其角色不同安装内容也会不一样。

- 系统环境安装（对所有主机）

这包含操作系统的安装、epel源的更新以及防火墙的重新设置。

(1) 系统安装

对所有主机安装CentOS 7.0。为了节省时间，建议选择“最小安装”，这样安装的CentOS没有图形界面。

(2) 添加epel源并更新

通过以下代码安装一些基础工具，例如wget、时钟同步的ntpdate、DNS工具包bind-utils和epel源：

```
# yum -y install wget ntpdate bind-utils
# wget http://mirror.centos.org/centos/7/extras/x86_64/Packages/epel-release-7-2.noarch.rpm
```

```
#yum install epel-release-7-2.noarch.rpm
# yum update
```

(3) 更改防火墙

CentOS 7.0默认使用firewall为防火墙，这里我们将其更改为iptables，具体步骤如下。

1) 停用firewall并禁止开机启动，其操作为：

```
# systemctl stop firewalld.service
# systemctl disable firewalld.service
```

2) 安装iptables，其操作为：

```
# yum install iptables-services #安装
# systemctl start iptables.service #重启使配置生效
# systemctl enable iptables.service #设置开机启动
```

- 安装和配置Etcd（192.168.1.84主机）

在192.168.1.84这台主机上安装Etcd，用于存储Kubernetes的动态信息。

(1) 获取和安装Etcd，其操作为：

```
# wget https://github.com/coreos/etcd/releases/download/v0.4.6/etcd-v0.4.6-linux-amd64.tar.gz
# tar -zxvf etcd-v0.4.6-linux-amd64.tar.gz
# cd etcd-v0.4.6-linux-amd64
# cp etcd* /bin/
# /bin/etcd -version
etcd version 0.4.6
```

(2) 启动Etcd服务，其操作为：

```
#mkdir /data/etcd
#/bin/etcd -name etcdserver -peer-addr 192.168.1.84:7001 -addr 192.168.1.84:4001 -data-dir /data/etcd
-peer-bind-addr 0.0.0.0:7001 -bind-addr 0.0.0.0:4001 &
```

参数-peer-addr指定与其他节点通信的地址，-addr指定服务监听地址，-data-dir指定数据存储目录。

(3) 配置Etcd的防火墙，允许4001和7001端口的请求：

```
# iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 4001 -j ACCEPT
# iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 7001 -j ACCEPT
```

安装Kubernetes（主控节点和从属节点）

不管是主控节点还是从属节点，安装Kubernetes的步骤是一致的，主要有3种方式，一是通过yum源，二是下载最新的二进制文件进行安装，三是通过编译源码然后进行安装。

通过yum源安装的操作如下：

```
# curl https://copr.fedoraproject.org/coprs/eparis/kubernetes-epel-7/repo/epel-7/eparis-kubern etes
```

```
-epel-7-epel-7.repo -o /etc/yum.repos.d/eparis-kubernetes-epel-7-epel-7.repo
#yum -y install kubernetes
```

这种方式安装的并不是最新版本，如果需要使用最新版本，则按第二种方法进行安装或者升级。

下载最新的二进制文件，安装最新版本。

可以访问https://github.com/GoogleCloudPlatform/kubernetes/releases查看当前最新的版本，下载下来之后解压，然后复制/bin下的二进制文件到/usr/bin目录中。例如，当前为v0.7.2版本，具体操作为：

```
#wget https://github.com/GoogleCloudPlatform/kubernetes/releases/download/v0.7.2/kuberne
    tes.tar.gz
#tar -zxvf kubernetes.tar.gz
# tar -zxvf kubernetes/server/kubernetes-server-linux-amd64.tar.gz
# cp kubernetes/server/bin/kube* /usr/bin
```

源码编译安装。

和第二种方法一样，可以通过https://github.com/GoogleCloudPlatform/kubernetes/releases下载最新的源代码。如果你不需要对Kubernetes进行开发，而只是将源码编译成二进制，就不需要配置golang环境。具体操作为：

```
#git clone https://github.com/GoogleCloudPlatform/kubernetes.git
#cd kubernetes
#make release
```

然后将相关二进制文件复制到/usr/bin中。如无特别需求，推荐使用第二种方法进行安装。

通过上述3种方式的任何一种安装完毕后，还要检查安装是否正确，这可以通过检测版本信息来验证，具体如下：

```
# /usr/bin/kubectl version
    Client Version: version.Info{Major:"0", Minor:"6+", GitVersion:"v0.6.2",
GitCommit:"729fde276613eedcd99ecf5b93f095b8deb64eb4", GitTreeState:"clean"}
Server Version: &version.Info{Major:"0", Minor:"6+", GitVersion:"v0.6.2",
GitCommit:"729fde276613eedcd99ecf5b93f095b8deb64eb4", GitTreeState:"clean"}
```

- 配置主控节点

前面说过，主控节点包含apiserver、调度器和控制器管理器这3个组件，相关配置也涉及这三块。

下面简要说明一下如何启动主控节点的3大组件，并将其加入开机启动中。

(1) /etc/kubernetes/config文件的配置，具体为：

```
# Comma seperated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.1.84:4001"
# logging to stderr means we get it in the systemd journal
```

```
KUBE_LOGTOSTDERR="--logtostderr=true"
# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"
# Should this cluster be allowed to run privleged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

上述代码配置了Etcd服务器的地址和端口、日志的输出方式和级别以及是否支持特权容器。

(2) /etc/kubernetes/apiserver文件的配置，具体为：

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"
# The port on the local server to listen on.
KUBE_API_PORT="--port=8080"
# How the replication controller and scheduler find the kube-apiserver
KUBE_MASTER="--master=192.168.1.83:8080"
# Port minions listen on
KUBELET_PORT="--kubelet_port=10250"
# Address range to use for services
KUBE_SERVICE_ADDRESSES="--portal_net=10.254.0.0/16"
# Add you own!
KUBE_API_ARGS=""
```

(3) /etc/kubernetes/controller-manager文件的配置，具体为：

```
# Comma seperated list of minions
KUBELET_ADDRESSES="--machines= 192.168.1.85,192.168.1.86"
# Add you own!
KUBE_CONTROLLER_MANAGER_ARGS=""
```

其中KUBELET_ADDRESSES列出了使用kubelet组件的从属节点。

(4)/etc/kubernetes/scheduler文件的配置为：

```
# Add your own!
KUBE_SCHEDULER_ARGS=""
```

(5) 启动主控节点的Kubernetes服务，具体操作为：

```
# systemctl daemon-reload
# systemctl start kube-apiserver.service kube-controller-manager.service kube-scheduler.service
# systemctl enable kube-apiserver.service kube-controller-manager.service kube-scheduler.service
```

- 配置从属节点

从属节点包含kubelet和服务代理这两个组件，下面介绍一下如何配置这两个组件。

(1) 编辑/etc/sysconfig/docker，以便后续提供远程API维护：

```
#vi /etc/sysconfig/docker
```

加入如下行：

```
OPTIONS=--selinux-enabled -H tcp://0.0.0.0:2376 -H fd://
```

(2) 修改从属节点的防火墙，以保证主控节点能够连接到它：

```
iptables -I INPUT -s 192.168.1.83 -p tcp --dport 10250 -j ACCEPT
```

(3) 修改/etc/kubernetes/config文件，具体内容为：

```
# Comma seperated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.1.84:4001"
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"
# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"
# Should this cluster be allowed to run privleged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

该文件配置了Etcd服务器的地址和端口、日志的输出方式和级别等信息。

(4) 修改/etc/kubernetes/kubelet文件：

```
###
# kubernetes kubelet (minion) config
# The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"
# The port for the info server to serve on
KUBELET_PORT="--port=10250"
# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname_override=192.168.1.85"
# Add your own!
KUBELET_ARGS=""
```

(5) 编辑/etc/kubernetes/proxy文件，具体内容为：

```
KUBE_PROXY_ARGS=""
```

(6) 启动Kubernetes服务，具体操作为：

```
# systemctl daemon-reload
# systemctl enable docker.service kubelet.service kube-proxy.service
# systemctl start docker.service kubelet.service kube-proxy.service
```

我们启动了docker、kubelet和kube-proxy服务，并且将它们加入到开机启动中。至此，我们已经将Kubernetes集群环境搭建完毕。

3. 验证安装

在这一节中，我们通过Kubernetes命令操作来验证Kubernetes集群是否成功搭建。要操作Kubernetes集群，需要在主控主机上操作，或者能访问主控主机8080端口的Kubernetes客户端主机上操作。

- 常用命令

Kubernetes的操作是通过Kubernetes的客户端连接到主控节点的apiserver来进行的。客户端可

以用命令行的方式进行操作，这主要使用kubectl工具实现。

- kubectl get minions：查看从属主机。
- kubectl get pods：查看Pod清单。
- kubectl get services 或kubectl get services -o json：查看service清单。
- kubectl get replicationControllers：查看备份控制器清单。
- for i inkubectl get pod|tail -n +2|awk '{print $1}'; do kubectl delete pod $i; done：删除所有Pod。

- REST方式

除了使用kubectl命令的形式外，还可以使用REST方式访问，这种方式比前者实时性更高。

- curl -s -L http://192.168.1.83:8080/api/v1beta1/version | python -mjson.tool：查看Kubernetes版本。
- curl -s -L http://192.168.1.83:8080/api/v1beta1/pods | python -mjson.tool：查看Pod清单。
- curl -s -L http://192.168.1.83:8080/api/v1beta1/replicationControllers | python -mjson.tool：查看备份控制器清单。
- curl -s -L http://192.168.1.83:8080/api/v1beta1/minions | python -m json.tool：查看从属主机。
- curl -s -L http://192.168.1.83:8080/api/v1beta1/services | python -m json.tool：查看service清单。

- 创建Pod单元

接下来，我们创建一个简单的Apache应用。首先，新建JSON配置脚本apache-pod.json：

```
{
    "id": "fedoraapache",
    "kind": "Pod",
    "apiVersion": "v1beta1",
    "desiredState": {
        "manifest": {
            "version": "v1beta1",
            "id": "fedoraapache",
            "containers": [{
                "name": "fedoraapache",
                "image": "fedora/apache",
                "ports": [{
                    "containerPort": 80,
                    "hostPort": 8080
                }]
            }]
        }
    },
    "labels": {
```

```
        "name": "fedoraapache"
    }
}
```

然后执行如下命令：

```
# kubectl create -f apache-pod.json
```

验证Pod是否创建，可以通过如下命令查看：

```
# kubectl get pod
NAME            IMAGE(S)          HOST              LABELS              STATUS
fedoraapache    fedora/apache     192.168.1.86/     name=fedoraapache   Running
```

这里我们配置了一个简单的apache容器，可以通过访问地址为192.168.1.86的从属节点的8080端口来访问，其效果图如图16-8所示。

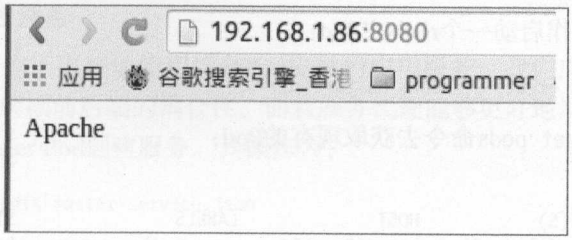


图16-8 使用装有Apache应用的容器创建Pod的简单效果图

16.4.2 应用操作

安装好Kubernetes的运行环境后，就可以在上面运行我们的应用服务了。为了更好地说明Kubernetes的操作，这里将采用Kubernetes官方提供的一个示例GuestBook来展示如何创建和操作Kubernetes集群，其中GuestBook的相关文件和说明在Kubernetes的examples/guestbook目录下。本节所有操作具有如下3个前提。

- Kubernetes环境已经按16.4.1节所示的步骤搭建成功。
- kubectl命令操作是在主控节点下进行的。
- 相关文件操作需要有Kubernetes官网提供的案例Guestbook，并在该案例目录下。

最后，我们将建立起如图16-9所示的架构。前端由3个php-redis Pod组成，它们由frontend备份控制器生成。后端创建了一个redis-master Pod和两个redisslave Pod，并为它们分别创建redis-master服务和redisslave服务。此外，redisslave Pod是由redisslave备份控制器创建的。

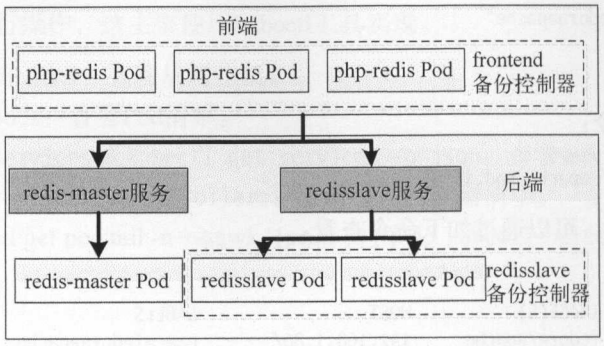


图16-9　GuestBook案例架构图

第1步：启动redis主Pod

首先，通过如下操作启动一个redis主Pod：

```
$ cluster/kubectl.sh create -f redis-master.json
```

接着使用kubectl get pods命令去获取现有的Pod：

```
# kubectl get pods
NAME           IMAGE(S)           HOST              LABELS              STATUS
redis-master   dockerfile/redis   192.168.1.86/     name=redis-master   Pending
```

注意STATUS一栏，这里显示为Pending（等待）。这是由于第一次运行时，需要从网络下载镜像来创建容器，所以需要等待。下载完毕之后，其状态就会转为Running。接下来，我们分析一下redis-master.json文件，其内容为：

```
{
    "id": "redis-master",
    "kind": "Pod",
    "apiVersion": "v1beta1",
    "desiredState": {
        "manifest": {
            "version": "v1beta1",
            "id": "redis-master",
            "containers": [{
                "name": "master",
                "image": "dockerfile/redis",
                "cpu": 100,
                "ports": [{
                    "containerPort": 6379,
                    "hostPort": 6379
                }]
            }]
        }
    },
    "labels": {
        "name": "redis-master"
```

```
    }
}
```

它是以JSON格式存储的，下面简要说明各个参数的含义。

- id：它是Pod、服务等各类资源的唯一标识字段。需要说明的是，在Kubernetes中，Pod、服务、控制器等都可以称作资源。
- kind：它是资源类型，可以是Pod、Service和ReplicationController。
- apiVersion：它目前是v1beta1。
- containers：字段跟容器相关，这里将其命名为master，它基于dockerfile/redis镜像构建，还将主机上的6379端口映射到容器的6379端口。
- labels标签：里面的键值对为"name"="redis-master"，这个用于后续服务。根据该标签可以决定是否使用该Pod提供服务。

第2步：启动redis主服务

由原理部分知道，为了更好地使用后台Pod提供的应用服务，我们应该为其创建Kubernetes的服务。使用服务能够解耦前后端的耦合性，而且服务代理能够更好地均衡网络负载。接下来，我们为第1步的redis-master Pod创建服务。其操作为：

```
$kubectl create -f redis-master-service.json
redis-master
```

创建成功后，可以通过下面的命令来查看服务状况：

```
$kubectl get services
NAME          LABELS           SELECTOR                          IP              PORT
kubernetes-ro <none>           component=apiserver,provider=kubernetes  10.254.18.26    80
kubernetes    <none>           component=apiserver,provider=kubernetes  10.254.47.133   443
redis-master name=redis-master name=redis-master                 10.254.193.174  6379
```

可以看到，有3项服务，其中kubernetes-ro和kubernetes是Kubernetes的内置核心服务，redis-master即为我们刚刚创建的服务。服务的PORT为10.254.193.174:6379。下面查看一下redis-master-service.json文件的内容，具体为：

```
{
    "id": "redis-master",
    "kind": "Service",
    "apiVersion": "v1beta1",
    "port": 6379,
    "containerPort": 6379,
    "selector": {
        "name": "redis-master"
    },
    "labels": {
        "name": "redis-master"
    }
}
```

其id为redis-master，资源类型为Service，port是服务端口，其值为6379，这意味着如果服务代理监听到6379端口有请求，则认为是对redis-master服务的请求。containerPort是容器的开放端口，这和Pod中的是一样的。此外，selector是选择子字段，labels是标签字段，它们都包含name=redis-master键值对。

第3步：启动redis的备份从属Pod

我们知道，备份控制器可以为某个Pod创建备份，并保证集群中运行的Pod数量始终为指定数量，这对于提供一个可增缩容的稳定的服务非常有必要。通过如下命令来创建备份控制器redisSlaveController：

```
$ kubectl create -f ./redis-slave-controller.json
redisSlaveController
```

然后通过以下命令查看该备份控制器：

```
kubectl get replicationcontrollers
NAME                    IMAGE(S)                    SELECTOR            REPLICAS
redisSlaveController    brendanburns/redis-slave    name=redisslave     2
```

它的镜像为brendanburns/redis-slave，选择子为name=redisslave，备份数为2。查看redis-slave-controller.json文件的内容：

```
{
    "id": "redisSlaveController",
    "kind": "ReplicationController",
    "apiVersion": "v1beta1",
    "desiredState": {
        "replicas": 2,
        "replicaSelector": {"name": "redisslave"},
        "podTemplate": {
            "desiredState": {
                "manifest": {
                    "version": "v1beta1",
                    "id": "redisSlaveController",
                    "containers": [{
                        "name": "slave",
                        "image": "brendanburns/redis-slave",
                        "cpu": 200,
                        "ports": [{"containerPort": 6379, "hostPort": 6380}]
                    }]
                }
            },
            "labels": {
                "name": "redisslave",
                "uses": "redis-master",
            }
        }},
    "labels": {"name": "redisslave"}
}
```

replicas字段是备份数量。ports字段定义了容器端口，因为这是由容器服务决定的，所以和Pod、Service一样，都为6379。这里hostPort为6380，这是因为备份控制器创建的是Pod，这些Pod会被分配到两台从属主机上，和redis-master Pod同时存在，如果使用相同的端口，则会发生冲突，导致后创建的Pod启动不了。标签字段有两个键值对，一个为name=redisslave，一个为uses=redis-master，表征它们是redis-master Pod的备份。

备份控制器创建好后，就可以查看当前集群中Pod的运行状况，具体如下：

```
[root@localhost guestbook]# kubectl get pods
NAME                                    IMAGE(S)                    HOST
LABELS                                  STATUS
87a7b08b-9a06-11e4-8384-080027a11a67    brendanburns/redis-slave    192.168.1.85/
name=redisslave,uses=redis-master       Running
redis-master                            dockerfile/redis            192.168.1.86/
name=redis-master                       Running
87a64d15-9a06-11e4-8384-080027a11a67    brendanburns/redis-slave    192.168.1.86/
name=redisslave,uses=redis-master       Running
```

可以看到，这里有一个redis-master Pod和两个redisslave从属Pod。

第4步：创建redis从属服务

和第2步一样，这里为redisslave Pod创建服务，具体操作为：

```
$kubectl create -f ./redis-slave-service.json
redisslave
$ kubectl get services
NAME            LABELS            SELECTOR                                IP                  PORT
kubernetes-ro   <none>            component=apiserver,provider=kubernetes 10.254.18.26        80
kubernetes      <none>            component=apiserver,provider=kubernetes 10.254.47.133       443
redis-master    name=redis-master name=redis-master                       10.254.193.174      6379
redisslave      name=redisslave   name=redisslave                         10.254.202.65       6379
```

可以看到，我们新创建了redisslave服务，其PORT为10.254.202.65:6379。

第5步：创建前端Pod

为了使用redis服务，我们创建PHP前端服务Pod，具体操作为：

```
$ cluster/kubectl.sh create -f examples/guestbook/frontend-controller.json
frontendController
```

这里没有使用创建Pod的脚本而是直接通过备份控制器来创建。查看备份控制器的具体情况：

```
[root@localhost guestbook]# kubectl get replicationcontrollers
NAME                    IMAGE(S)                    SELECTOR            REPLICAS
redisSlaveController    brendanburns/redis-slave    name=redisslave     2
frontendController      brendanburns/php-redis      name=frontend       3
```

可以看到刚创建的frontendController。查看一下Pod的具体情况，具体操作为：

```
$kubectl get pods
```

```
NAME                                    IMAGE(S)                HOST
LABELS                                  STATUS
87a7b08b-9a06-11e4-8384-080027a11a67    brendanburns/redis-slave    192.168.1.85/
name=redisslave,uses=redis-master       Running
8429cb75-9a14-11e4-8384-080027a11a67    brendanburns/php-redis      192.168.1.85/
name=frontend,uses=redisslave,redis-master  Running
redis-master                            dockerfile/redis            192.168.1.86/
name=redis-master                       Running
87a64d15-9a06-11e4-8384-080027a11a67    brendanburns/redis-slave    192.168.1.86/
name=redisslave,uses=redis-master       Running
8428b3ec-9a14-11e4-8384-080027a11a67    brendanburns/php-redis      192.168.1.86/
name=frontend,uses=redisslave,redis-master  Running
842c6bee-9a14-11e4-8384-080027a11a67    brendanburns/php-redis      <unassigned>
name=frontend,uses=redisslave,redis-master  Pending
```

可以发现，这里有一个redis-master Pod、两个redisslave节点和3个前端Pod。在3个前端Pod中，有一个的HOST字段为<unassigned>，其状态也是Pending，这是为什么呢？我们来看一下frontendcontroller.json文件：

```
{
    "id": "frontendController",
    "kind": "ReplicationController",
    "apiVersion": "v1beta1",
    "desiredState": {
        "replicas": 3,
        "replicaSelector": {"name": "frontend"},
        "podTemplate": {
            "desiredState": {
                "manifest": {
                    "version": "v1beta1",
                    "id": "frontendController",
                    "containers": [{
                        "name": "php-redis",
                        "image": "kubernetes/example-guestbook-php-redis",
                        "cpu": 100,
                        "memory": 50000000,
                        "ports": [{"containerPort": 80, "hostPort": 8000}]
                    }]
                }
            },
            "labels": {
                "name": "frontend",
                "uses": "redisslave,redis-master"
            }
        }},
    "labels": {"name": "frontend"}
}
```

可以看到，replicas字段为3，即创建3个前端Pod，而我们的集群只有两个从属节点，这势必会让两个前端Pod分配到一台机器上。而在Pod的定义中，有一项为hostPort，这是Pod要使用的从属节点上的宿主端口，这里为8000端口，它一旦被某个Pod使用，就不能再分配给其他Pod。

这导致3个Pod中必有一个Pod会因为分配不到端口而创建失败。

第6步：访问前端服务

至此，我们已经在Kubernetes集群中搭建完PHP+redis服务，通过访问http:// 192.168.1.85:8000或者http:// 192.168.1.86:8000端口来使用该服务，如图16-10所示。

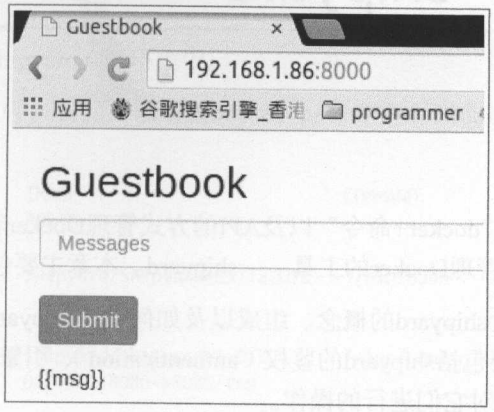


图16-10 GuestBook示例页面

第 17 章 shipyard

前面我们通过命令行“docker+命令”以及API的方式管理Docker容器，在这一章中，我们将介绍一款通过Web UI方式管理Docker的工具——shipyard。本章主要包含如下内容：

- shipyard简介，包含shipyard的概念、组成以及如何安装shipyard；
- shipyard操作，主要包括shipyard的鉴权（authentication）、引擎、容器等概念以及通过API和Web UI两种方式对它们进行的操作。

17.1 简介

shipyard是一款Docker管理工具，基于Docker集群管理工具包Citadel，提供Web UI、命令行CLI以及API三种方式管理集群。shipyard能够管理Docker集群中的容器、主机等各种资源。shipyard的核心功能是管理Docker容器，然而如果使用Extension Images，你还可以实现应用路由、负载均衡、日志集中化以及应用部署等。本章主要说明如何使用shipyard来管理Docker集群。

shipyard提供如下三种方式来管理Docker集群的资源。

- **Web UI**。通过Web UI，可以进行简单的Docker集群管理，查看集群的资源使用率、引擎的使用率，创建和销毁容器，查看集群范围内的事件等。
- **命令行CLI**。shipyard也提供了强大的命令行接口，实现了其API，通过CLI能够发挥出其全部功能。
- **API**。shipyard的核心是API，shipyard的CLI和Web UI都是通过API来实现其功能的，所以API是基石。通过使用服务密钥，你可以使用API来管理Docker集群。

接下来，我们说明如何安装shipyard。

shipyard使用RethinkDB存储账号、引擎、服务密钥和扩展元数据等信息。所以，先通过如下操作配置好RethinkDB容器：

```
$ docker run -it -d --name shipyard-rethinkdb-data   --entrypoint /bin/bash shipyard/rethinkdb -l
$ docker run -it -P -d --name shipyard-rethinkdb --volumes-from shipyard-rethinkdb-data
shipyard/rethinkdb
```

然后下载和启动shipyard容器：

```
$ docker run -it -p 8080:8080 -d --name shipyard --link shipyard-rethinkdb:rethinkdb shipyard/shipyard
```

为了能够通过CLI来访问shipyard，我们还需要安装shipyard的客户端，具体操作为：

```
$ docker run -ti -v /Users/root/.boot2docker:/b2d --rm shipyard/shipyard-cli
shipyard cli>
```

运行该容器，即进入shipyard CLI环境中。

安装好上述4个容器之后，我们可以通过docker ps命令查看当前运行的容器：

```
$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED
STATUS              PORTS                                                   NAMES
3df3ccfb9ae4        shipyard/shipyard-cli:2.0.8    "/bin/bash"              41 hours ago
Up 41 hours                                                                 romantic_rosalind
586359c2b033        shipyard/shipyard:2.0.8        "/app/controller"        2 days ago
Up 2 days           0.0.0.0:8080->8080/tcp                                  shipyard
ecba7de12a9a        shipyard/rethinkdb:latest      "/usr/bin/rethinkdb      2 days ago
Up 2 days           0.0.0.0:49153->28015/tcp, 0.0.0.0:49154->29015/tcp,
0.0.0.0:49155->8080/tcp  shipyard-rethinkdb
74b789abaefb        shipyard/rethinkdb:latest      "/bin/bash -l"           2 days ago
Up 2 days           28015/tcp, 29015/tcp, 8080/tcp                          shipyard-rethinkdb-data
```

可以看到，shipyard的4个容器已经跑起来了。shipyard监听本地的8080端口。

接下来，我们可以登录shipyard，具体有两种方式，一种是Web UI方式，另一种是CLI方式。shipyard提供一个默认的管理员账号，用户名为admin，密码为shipyard。相对于CLI，Web UI更为直观、简单，但其功能也没有CLI强大。首先，通过浏览器访问本地的8080端口，如图17-1所示。

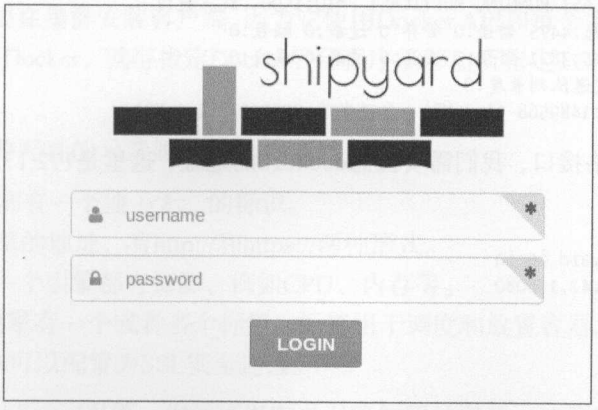


图17-1 shipyard登录页面

输入账号和密码，即可登录到管理界面，如图17-2所示。

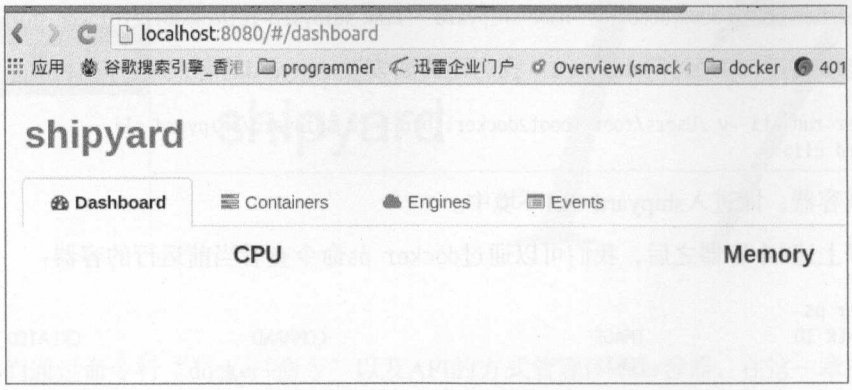


图17-2　shipyard管理主页面

现在我们再说明一下CLI方式。当运行docker run -ti -v /Users/root/.boot2docker:/b2d --rm shipyard/shipyard-cli命令时，终端就进入shipyard客户端的操作界面。我们通过shipyard login命令登录，具体为：

```
shipyard cli> shipyard login
URL: http://localhost:8080
Username: admin
Password:
FATA[0015] Post http://localhost:8080/auth/login: dial tcp 127.0.0.1:8080: connection refused
```

提示出错，连接被拒绝。我们通过ifconfig命令来查看当前主机的网络接口信息：

```
$ ifconfig
docker0   Link encap:以太网  硬件地址 56:84:7a:fe:97:99
          inet 地址:172.17.42.1  广播:0.0.0.0  掩码:255.255.0.0
          inet6 地址: fe80::5484:7aff:fefe:9799/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:4475 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:3821 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:1489568 (1.4 MB)  发送字节:2052137 (2.0 MB)
```

主机会有很多网络接口，我们需要注意docker0的地址，这里是172.17.42.1。我们使用该地址登录，具体为：

```
shipyard cli> shipyard login
URL: http://172.17.42.1:8080
Username: admin
Password:
shipyard cli>
```

登录成功。

17.2 shipyard 操作

在这一节中，我们会介绍shipyard的基础概念及相关操作。

17.2.1 鉴权

想要操作shipyard管理的Docker集群，就必须通过账号登录。shipyard支持多用户，你可以创建多个账号，方便多人来管理。当然，只有管理员有添加账号的权限。账号分两种角色——admin和user，前者拥有集群的所有管理权限，后者可以进行账号、服务密钥外的任何管理操作。

要想创建一个新账号，可以通过shipyard add-account命令完成，具体操作为：

```
shipyard cli> shipyard add-account -u minimicall -p 110110 -r admin
```

其中参数-u用于设置用户名，-p用于设置密码，-r则用于设置角色，这里我们添加了一个管理员。添加好账号后，可以通过如下命令查看shipyard中有哪些账号：

```
shipyard cli> shipyard accounts
Username Role
admin  admin
minimicall admin
```

可以看到，当前有两个管理员：admin和minimicall。

另外，如果想删除某个账号，可以通过delete-account命令来完成，具体如下：

```
shipyard cli> shipyard delete-account minimicall
```

17.2.2 引擎

shipyard集群可以包含一个或多个引擎（engine）。引擎就是一个在后台监听TCP的Docker守候进程。shipyard不需要在集群安装客户端，因为它使用Docker API和每个主机的Docker后台通信。想要通过shipyard管理Docker，就得指定Docker后台程序能够监听指定TCP端口，这在14.2节中已经介绍过了。

添加引擎时，需要配置的元素如下所示。

- id：每个引擎都有一个独一无二的标识。
- addr：它是引擎的地址，有http://和https://两种形式。
- resource：每一个引擎都有资源，例如CPU、内存等。
- label：一个引擎有一个或者多个标签，标签用于调度和放置容器。
- ssl：一个引擎可以配置为SSL安全连接。

一旦添加了一个shipyard引擎，你就可以定义引擎使用的资源。这些限制在容器的调度时非

常有用，以保证能够满足用户的需求。你也可以设定SSL证书，提供安全的通信。

接下来，我们说明引擎的相关操作示例。

添加引擎的代码如下：

```
shipyard cli> shipyard add-engine --id local \
    --addr http:// 172.17.42.1:2376 \
    --cpus 4.0 \
    --memory 8192 \
    --label dev \
    --label local
```

上述代码添加了一个id为local的引擎，addr指定了该Docker后台监听的地址和端口，cpus表示最多使用4个CPU，memory用于设置内存上限，label参数设定了dev和local两个标签。

查看引擎的代码如下：

```
shipyard cli> shipyard engines
ID      Cpus    Memory  Host                    Labels
local   4.00    8192.00 http:// 172.17.42.1:2376 local,dev
```

在上述代码中，可以看到我们刚刚添加的引擎。

查看引擎详细信息的代码如下：

```
shipyard cli> shipyard inspect-engine local
{
    "engine": {
        "labels": [
            "local",
            "dev"
        ],
        "memory": 2048,
        "cpus": 4,
        "addr": "172.17.42.1:2376",
        "id": "local"
    },
    "id": "a08b8518-e963-4eb5-959a-566bd270cd28"
}
```

删除引擎的代码如下：

```
shipyard cli> shipyard remove-engine a08b8518-e963-4eb5-959a-566bd270cd28
removed local
```

我们不仅可以通过shipyard cli命令对引擎进行增、删、改、查，还可以通过Web UI来进行这些操作。

添加引擎的界面如图17-3所示，从中点击ADD按钮，进入添加引擎配置页面，如图17-4所示。

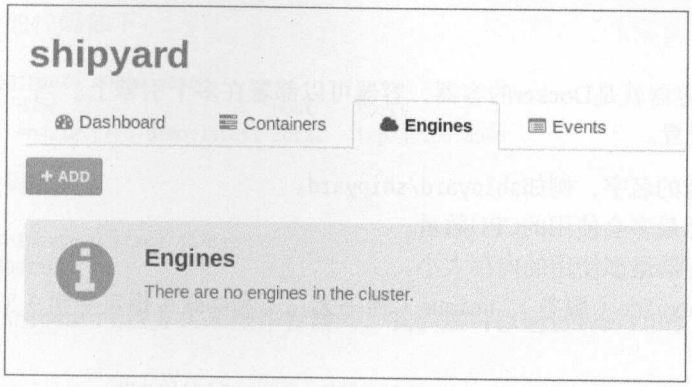


图17-3 添加引擎界面

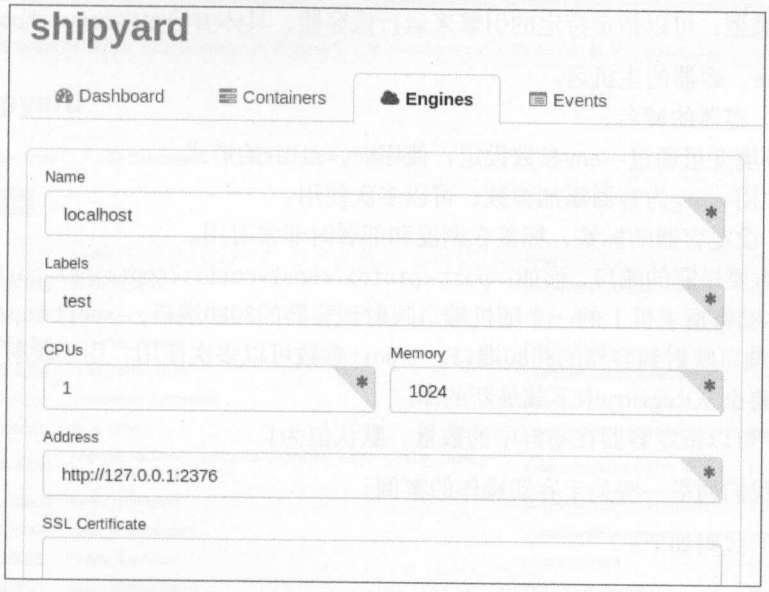


图17-4 添加引擎配置页面

配置好之后，点击“提交”按钮。添加引擎成功后的界面，如图17-5所示。

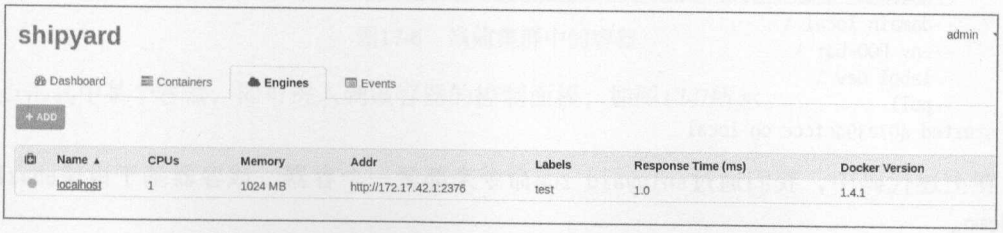


图17-5 添加引擎成功

17.2.3　容器

shipyard中的容器就是Docker的容器，容器可以部署在多个引擎上。当一个容器被部署时，有以下参数可以配置。

- Name。镜像的名字，例如shipyard/shipyard。
- CPUs。容器最多会使用的CPU数量。
- Memory。容器最多使用的内存大小。
- Type。有service（服务）、unique（独一无二）和host（指定主机）3种类型，具体如下所示。
  - service**类型**：只有拥有指定标签的引擎才可以运行该容器。
  - unique**类型**：只有该引擎上没有相同的容器运行时，容器才会被调度到该引擎。
  - host**类型**：可以指定特定的引擎来运行该容器，具体方法是--label host:<host-id>。
- Hostname。容器的主机名。
- Domain。容器的域名。
- Env。环境变量通过--env参数设定，使用key=value的形式。
- Arg。使用--arg为容器添加参数，可以多次使用。
- Label。设定容器的标签，标签在调度和部署时非常有用。
- Port。需要暴露的端口，形如--port <proto>/<host-port>:<container-port>，例如--port tcp/:8080将宿主机上的一个随机端口映射到容器的**8080**端口，--port tcp/80:8080将宿主机的**80**端口映射到容器的**8080**端口。--port参数可以多次使用，用于映射多个端口。
- Pull。将会从Registry中下载最新的库。
- Count。可以指定容器在集群中的数量，默认值为1。

接下来，我们列举一些关于容器操作的案例。

部署容器的代码如下：

```
shipyard cli> shipyard run --name ehazlett/go-demo \
    --cpus 0.1 \
    --memory 32 \
    --type service \
    --hostname demo-test \
    --domain local \
    --env FOO=bar \
    --label dev \
    --pull
started 407e39dc1ccc on local
```

在上述代码中，我们通过shipyard run命令来部署一个容器，该容器基于镜像ehazlett/go-demo。

查看容器列表的代码如下：

```
shipyard cli> shipyard containers
ID              Name                    Host    Ports
407e39dc1ccc    ehazlett/go-demo:latest local   tcp/49166:8080
```

销毁容器的代码如下：

```
shipyard cli> shipyard destroy 407e39
destroyed 407e39dc1ccc
```

除了可以通过shipyard cli方式来操作容器外，我们还可以通过Web UI的方式，下面简要介绍一下。

查看当前集群中的容器，如图17-6所示。

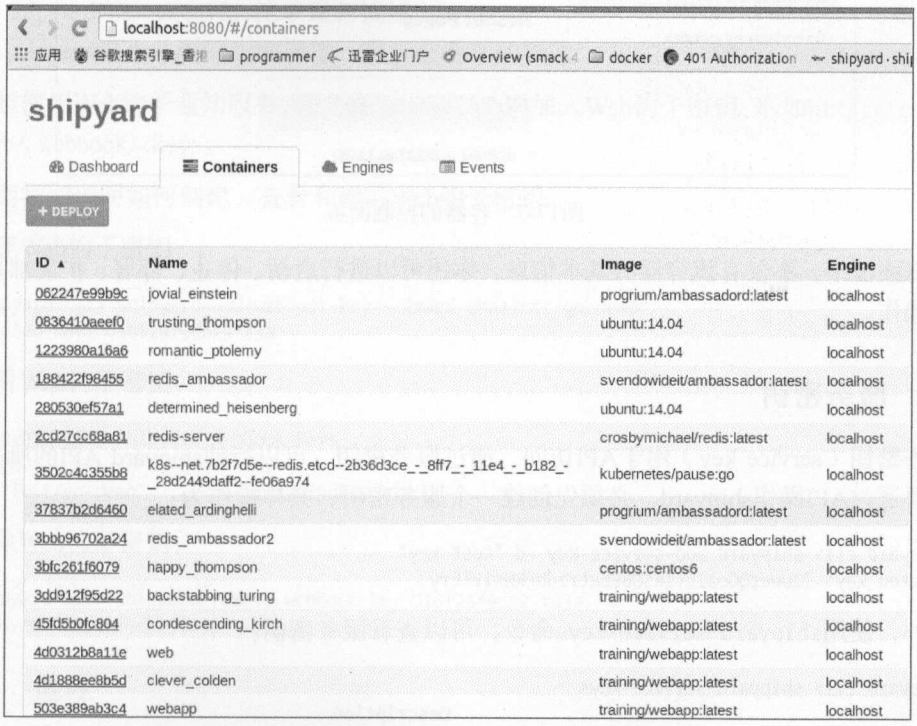


图17-6 当前集群中的容器

点击其中某个容器，即可进入到该容器的控制面板，如图17-7所示。

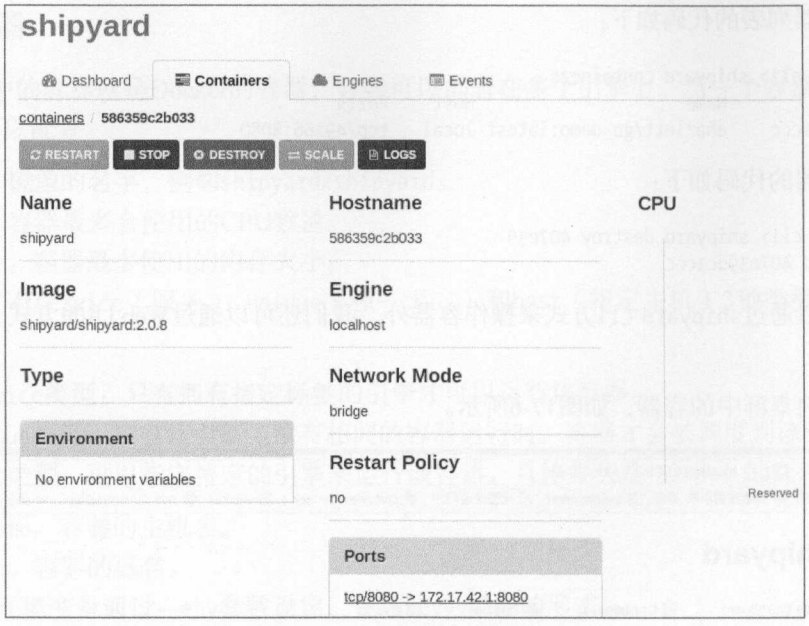


图17-7　容器的控制面板

在该面板中，不仅有该容器的基本信息，你还可以进行启动、停止、部署、扩展部署和查看日志等操作。

17.2.4　服务密钥

服务密钥（service key）用于API访问。通过服务密钥，可以行使shipyard API的所有权限。如果想要通过API操作shipyard，必须先创建一个服务密钥，具体操作为：

```
shipyard cli> shipyard add-service-key -d "test key"
created key: Z2uwezQGoaIcfiRSQBRbktrzdbFRWKlVTEry
```

此外，使用shipyard service-keys命令，可以查看服务密钥：

```
shipyard cli> shipyard service-keys
Key                                          Description
Z2uwezQGoaIcfiRSQBRbktrzdbFRWKlVTEry         test key
```

创建好密钥之后，我们可以通过该密钥以API的形式来访问shipyard的服务。和第9章学习Docker的API一样，这里我们使用curl命令来操作。例如，使用curl命令查询当前集群中的引擎：

```
 curl -s -H 'X-Service-Key: LdFRei6UpOnIcitk.dhHF/Fd7AgOQ87SWuVG' http://172.17.42.1:8080/api/engines
[{"id":"784f6429-2ce7-4f84-8296-30c79315fcc3","engine":{"id":"localhost","addr":"http://172.17.42
    .1:2376","cpus":1,"memory":1024,"labels":["test"]},"health":{"status":"up","response_time":918378},
    "docker_version":"1.4.1"}]
```

当不需要该密钥时，可以通过如下操作删除它：

```
shipyard cli> shipyard remove-service-key Z2uwezQGoaIcfiRSQBRbktrzdbFRWKlVTEry
removed Z2uwezQGoaIcfiRSQBRbktrzdbFRWKlVTEry
```

17.2.5 Web 钩子密钥

Web钩子密钥（webhook keys）用于Docker Hub与shipyard之间的通信，当Docker Hub中的镜像更新后，可以通知shipyard及时下载，具体流程如下所示。

(1) Docker Hub接到通知，创建一个新的镜像。

(2) Docker Hub向shipyard发送一个Web钩子通知。

(3) shipyard通过Web钩子密钥进行鉴权。

(4) shipyard向Docker Hub拉取最新镜像。

(5) shipyard停止并删除当前镜像，然后部署新的镜像。

若想使用Web钩子通知服务，需要在Docker Hub中加入Web钩子密钥，形如`http://<shipyard-url>/hub/ webhook/<key>`。

下面我们说明如何创建、查看和删除Web钩子密钥。

创建Web钩子密钥

```
shipyard cli> shipyard add-webhook-key --image ehazlett/go-demo
created key: 010f2af9db29f43a
```

查看Web钩子密钥：

```
shipyard cli> shipyard webhook-keys
Image                    Key
ehazlett/go-demo         010f2af9db29f43a
```

删除Web钩子密钥：

```
shipyard cli> shipyard remove-webhook-key 010f2af9db29f43a
removed 010f2af9db29f43a
```

17.2.6 事件

shipyard记录集群中的所有事件，例如容器的创建、启动、停止、密钥的管理、引擎的管理等。例如，可以通过shipyard events来查看系统中发生的所有事件：

```
shipyard cli> shipyard events
Time                     Message          Engine  Type     Tags
Sep 09 06:58:13 2014     container:6c07   local   start    docker
Sep 09 06:58:13 2014     container:6c07   local   create   docker
```

当然，通过Web UI也可以查看集群中发生的事件，如图17-8所示。

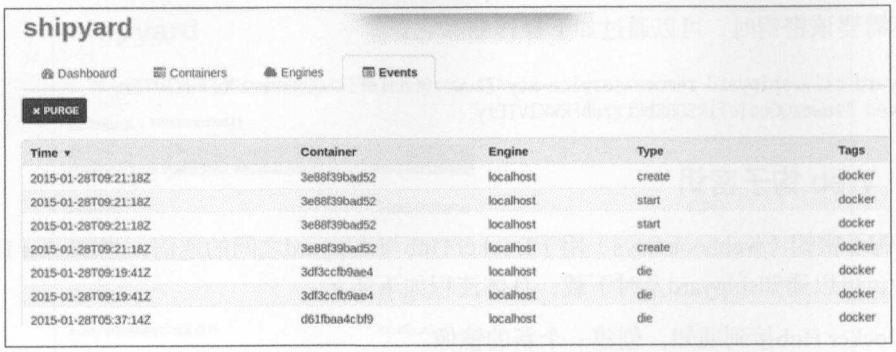


图17-8　shipyard集群中的事件记录

17.2.7　集群信息

shipyard还提供关于集群状态的信息，例如CPU、内存、容器、镜像、引擎以及尚未使用的CPU和内存等信息，具体的CLI操作为：

```
shipyard cli> shipyard info
Cpus: 4.00
Memory: 8192.00 MB
Containers: 2
Images: 5
Engines: 1
Reserved Cpus: 4.00% (0.16)
Reserved Memory: 3.52% (288.00 MB)
```

如果是Web UI，则在主控制面板中就可以看到集群的相关信息，如图17-9所示。

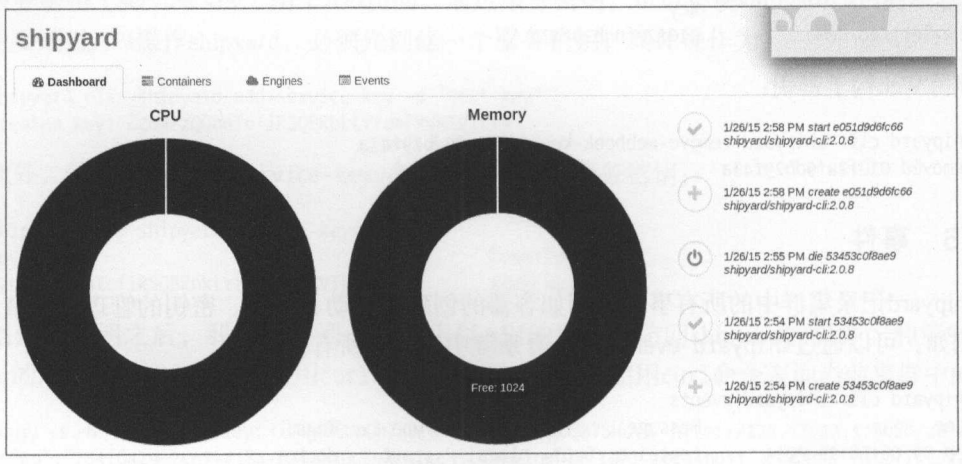


图17-9　shipyard集群信息

第18章 Machine+Swarm+Compose

在2014年12月份的DockerCon大会上，Docker给大家带来它的最新编排服务三件套：Machine、Swarm和Compose。Machine主要解决在各个平台上安装Docker的问题，让用户通过一条指令就可以安装好Docker，而无需知晓各个平台的具体安装方法。Swarm则是将Docker本地集群集中管理，让管理集群就像管理单个主机一样简单。Compose则用于安排应用部署到哪个容器组中。这三个组件都尚在开发之中，并没有正式的版本发布，所以不推荐大家在实际生产环境中尝试它们。在这一章中，我们要介绍的内容如下：

- Machine。包含Machine简介、安装和操作。
- Swarm简介。包含Swarm简介、原理和流程。
- Swarm实操。
- Swarm的发现服务和调度。
- Compose简介。

18.1 Machine

在1.4节中，我们介绍了如何在各个操作系统（如Ubuntu、Red Hat、OS X以及Windows）下安装Docker，然而在不同操作系统下安装的步骤相差颇大，并且我们还没有考虑另外一种情况，那就是在公共云平台下Docker的安装，此时我们需要先安装操作系统，然后再安装Docker，这个过程更漫长。程序员并不喜欢处理这种繁杂的事情，他们宁愿专注地写代码，所以急切渴望有一种工具能够让我们通过一条命令就可以安装好Docker，而不管它所在的环境。为了解决这个问题，Docker官方在2014年的DockerCon大会上提出了Machine。目前，Machine正在快速开发当中，尚未有正式发布的版本，其地址为https://github.com/docker/machine。

Machine是如何让用户无需关心平台的底层细节的呢？相信大家对“驱动”一词并不陌生，我们的操作系统中就有着各种各样的设备驱动。为了跨越各种硬件设备，操作系统向底层设备开放驱动API，功能的具体实现由硬件厂商来实现。Machine也是如此，其后端对Docker生态圈的各个平台开放API，只要你遵循API接口实现指定功能，那么Docker就可以在你的平台上跑起来。目前，已有的驱动有VirtualBox、Amazon Web Services、Google Compute Engine、Microsoft Azure以及

VMware vCloud Air等。

下面我们简单说明Machine的安装和操作。需要说明的是，Machine现在尚处在beta版本，我们不赞成读者将其用到真实的生产环境中。

要安装Machine，可以到https://docs.docker.com/machine/下载对应的二进制文件，然后将其改名为docker-machine，并将其放置到合适的目录下，接着将该目录加入到环境变量PATH中即可。

通过下面的命令验证一下：

```
$ docker-machine -v
machine version 0.1.0
```

发现一切正常。

Machine已经安装好了，接下来我们以VirtualBox环境为例，说明Machine的用法。为了在VirtualBox中使用Docker，首先要在你的系统上安装好VirtualBox，其版本应该为4.3.20或者更高。

在安装VirtualBox之前，可以通过docker-machine ls命令查看当前已有的主机实例：

```
$ docker-machine ls
NAME   ACTIVE   DRIVER   STATE   URL
```

可以发现，现在还没有实例。这里我们通过下面的命令安装VirtualBox下的Docker实例：

```
$ docker-machine create --driver virtualbox dev
INFO[0000] Creating SSH key...
INFO[0000] Creating VirtualBox VM...
INFO[0007] Starting VirtualBox VM...
INFO[0007] Waiting for VM to start...
INFO[0038] "dev" has been created and is now the active machine
INFO[0038] To connect: docker $(docker-machine config dev) ps
```

参数--driver指定驱动类型，这里是virtualbox，dev为创建的实例名称。当第一次运行该命令时，Machine会下载一个boot2docker.iso文件，它是一个包含Docker服务的轻量级Linux系统镜像文件。然后Machine会创建SSH密钥，创建VirtualBox VM并启动它。接着，我们再次查看已有主机实例：

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL
dev       *        virtualbox   Running   tcp:// 192.168.1.85.100:2376
```

可以发现，已经有一个名为dev的主机实例，其状态为Running，表示它目前是活动状态。URL则是其Docker守候进程的地址。

接下来，我们可以在该Docker服务上进行各种容器操作，例如：

```
$ docker $(docker-machine config dev) run busybox echo hello world
Unable to find image 'busybox' locally
Pulling repository busybox
```

```
e72ac664f4f0: Download complete
511136ea3c5a: Download complete
df7546f9f060: Download complete
e433a6c5b276: Download complete
hello world
```

在docker命令后面加入docker-machine config dev，指定命令作用于刚刚建立的dev主机实例。如果觉得每次输入这个config命令太麻烦，可以将dev实例的环境变量导入到当前系统，具体为：

```
$ $(docker-machine env dev)
```

dev实例的环境变量的内容为：

```
$ docker-machine env dev
export DOCKER_TLS_VERIFY=yes
export DOCKER_CERT_PATH=/home/root/.docker/machines/.client
export DOCKER_HOST=tcp:// 192.168.1.85:2376
```

设置好环境变量后，Docker默认就连接到dev主机实例的Docker后台，所以可以直接使用Docker命令：

```
$ docker run busybox echo hello world
hello world
```

有了Machine的主机实例后，我们可以通过下面的命令停止和启动该实例：

```
$ docker-machine stop dev
$ docker-machine start dev
```

关于Machine的内容，我们就介绍到这里。在这一节中，我们简单介绍了Machine的设计初衷以及基本操作，若想更加深入地了解它，可以访问GitHub上它的官网。

18.2 Swarm

Swarm是Docker公司于2014年12月初推出的一款Docker集群管理工具，其目标是使多台机器的Docker集群像只有一台机器的Docker一样容易管理。Swarm采用的是标准的Docker API作为其前端访问接口，这就意味着凡是使用Docker API进行通信的Docker客户端都可以透明地使用Swarm，例如dokku、Fig、krane、flynn、shipyard以及Docker本身的客户端，即二进制文件docker等。Swarm守护进程是使用Go语言编写的，截止到目前仍然处于Alpha阶段，然而其迭代速度却非常快，我们相信在不久的将来就可以将其投入到生产环境中。

作为Docker官方开发的工具，Swarm延续了Docker惯有的“batteries included but removable”设计理念。关于这个设计理念，可以这么理解：一方面，它属于Swarm集群中的一个重要组件，和Docker后台进程相互协作，使整个集群正常运行；另一方面，Swarm又是可插拔的，至少Docker后台完全不需依赖Swarm。此外，Swarm的内部组件也是可替换的，用户可以很方便地定制特定

的资源分配策略、调度算法等。

下面我们简要说明Swarm的组成、架构、流程以及原理等。

18.2.1　架构和组件

图18-1描述的是Swarm集群的架构图，主要包含Docker客户端、Swarm服务器（主控节点）、后端服务器、集群节点。Swarm接受来自Docker客户端的管理请求，它们之间采用Docker API联系。Docker客户端就像管理一台机器上的容器一样管理Swarm集群内的容器，Swarm接受到来自Docker客户端的命令之后，根据特定的过滤条件和调度算法，将该命令指派到指定节点运行。Swarm集群中的节点通过发现服务加入到Swarm集群中。此外，发现服务和响应的后端服务器一起维护管理着节点的健康状况、加入和退出等。

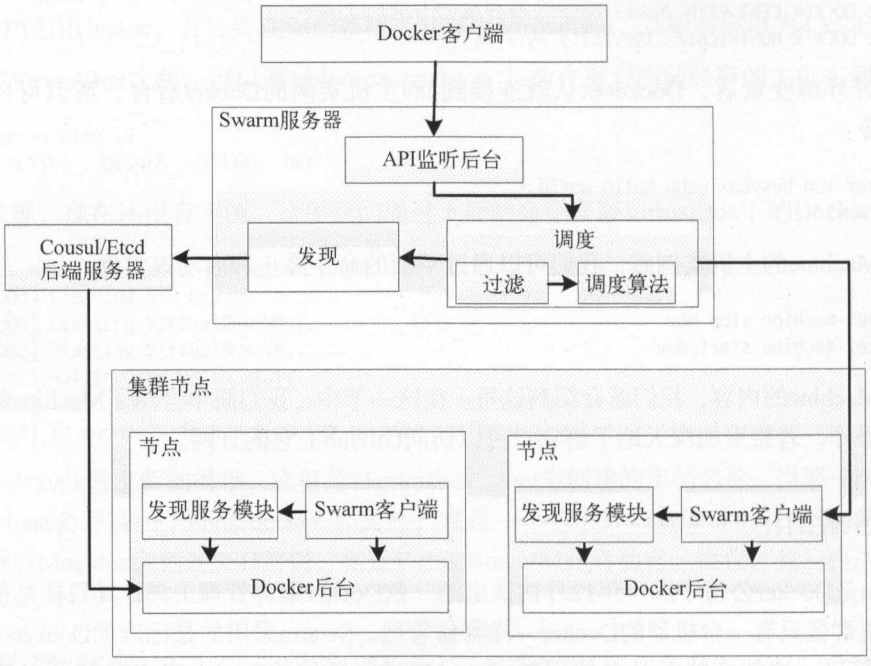


图18-1　Swarm集群架构图

Swarm和Docker客户端的联系与单节点的Docker命令并无区别。因为Swarm的监听端口通常也是2375和2376，用户可以通过-H IP:port的形式来连接即可，具体如下：

```
$ docker -H tcp://<swarm_ip:swarm_port> info
```

需要注意的是，Docker版本需高于1.4.0。这种形式和我们往常访问一台主机上的Docker后台并无区别，只是将IP和端口指向Swarm后台的IP和端口。然而，需要注意的是，由于Swarm和Docker

后台在结构上的差异，以及当下Swarm并不成熟，导致了许多命令在Swarm中不能使用，但我们相信这仅仅是时间问题。目前，可以使用的常见命令有：

- docker run
- docker create
- docker inspect
- docker kill
- docker logs
- docker start

在Swarm服务内部，主要包含发现服务和调度两大模块。

- **发现服务**。发现服务是Swarm用来维护集群状态的一种机制。集群中的节点会将自己的资源使用情况、健康状态等信息发送到Swarm节点或是诸如Consul、etcd等后端服务程序中，这些后端服务程序维护着节点列表以及每个节点的状态信息，Swarm通过查询后端服务器，得到每个节点的信息并跟踪集群中的节点，利用这些信息为调度模块提供决策支持。目前，Swarm提供了5种发现机制：节点发现（Node Discovery）、文件发现（File Discovery）、Consul发现（Consul Discovery）、Etcd发现（Etcd Discovery）和ZooKeeper发现（ZooKeeper Discover）。
- **调度**。调度模块负责命令的调度，通过标签过滤和调度算法，将客户端的命令指派给特定节点运行。在节点的Docker后台启动时，可以设定自己的一系列标签，这些标签可以是CPU数量、内存大小、端口等信息。当Swarm的调度模块接收到容器命令时，它首先根据标签过滤出一组符合条件的节点，然后在这组节点的基础上执行调度算法。目前，调度算法主要有装箱算法和随机算法。
- **Swarm节点**。Swarm集群中的节点包含Docker后台、Swarm客户端以及发现服务客户端。Docker后台就不多说了，Swarm客户端是负责和Swarm主控节点通信的模块，负责收发信息。发现服务客户端主要是和Consul、etcd进行通信的客户端。Swarm节点是任务的真正执行主体。

在说完Swarm的各个组件之后，我们现在说明其执行流程，具体如下所示。

(1) Swarm发现集群中的节点，收集集群中各个节点的状态、角色等信息并持续跟踪其状态，这主要通过发现服务模块以及相应的后端服务程序完成。

(2) 调度执行。有了第(1)步的发现，Swarm就知道集群中的节点资源、状态、角色等信息，并根据客户端投递过来的命令，通过一定的条件过滤和调度算法，选出目标节点。将命令推送到指定节点后，节点上的Swarm客户端接收该消息，并进一步将命令传递给节点的Docker后台程序，等待命令执行完毕后将相关信息返回给Swarm服务器。

(3) API监听。Swarm初始化好API监听接口后，就可以像Docker后台一样监听来自Docker客户端的命令，并将命令交给调度模块进行调度。

18.2.2　实操

由架构部分知道，Swarm集群主要包含一个主控节点和多个Swarm集群节点。创建一个Swarm集群并不难，你可以直接在现有的Docker集群上建立。特别是采用基于文件和节点的发现机制时，除了需要部署Swarm主控节点外，并不需要额外的配置。你甚至可以在现有的Docker节点上直接运行Swarm主控节点，不过我们建议在单独一台机器上运行它，因为它会处理较多的TCP连接，耗费比较多的文件描述符。接下来，我们说明Swarm的实际操作，其中所有节点的操作系统都为ubuntu:14.04。

1. 安装

获取Swarm的方式有两种，具体如下所示。

- 直接从Docker Hub中拉取Swarm镜像，其操作为：

```
$sudo docker pull swarm
$ docker images
REPOSITORY    TAG         IMAGE ID          CREATED           VIRTUAL SIZE
swarm         latest      92d78d321ff2      18 hours ago      7.19 MB
```

验证Swarm的版本：

```
$ docker run --rm swarm --version
swarm version 0.1.0 (a445ed9)
```

- 通过源码编译的方式。首先，你需要安装golang来构建Go语言编译环境：想要运行Swarm节点，必须保证Docker后台是1.4.0或者更新版本，并且所有的Docker后台是通过配置TCP连接的方式启动的。

```
$ sudo apt-get install -y golang git
```

然后创建一个目录：

```
mkdir go
```

接着将该目录加入到GOPATH环境变量中：

```
export GOPATH=~/go
```

再获取Swarm二进制文件，它将会自动下载到GOPATH变量表示的目录下：

```
go get -u github.com/docker/swarm
```

然后将Swarm二进制文件所在的目录添加进PATH变量中：

```
export PATH=$HOME/go/bin:$PATH
```

现在就可以直接使用Swarm了。下面验证其版本：

```
$ swarm --version
```

```
    swarm version 0.1.0 (a445ed9)
```

如果已经运行了Docker后台，则需要通过如下命令先停止它：

```
$ sudo service docker stop
```

然后通过下面命令将其启动：

```
$ sudo docker -H tcp://0.0.0.0:2375 -d &
2015/01/13 11:46:45 docker daemon: 1.0.1 990021a; execdriver: native; graphdriver:
[0189987e] +job serveapi(tcp://0.0.0.0:2375)
[0189987e] +job initserver()
[0189987e.initserver()] Creating server
2015/01/13 11:46:45 Listening for HTTP on tcp (0.0.0.0:2375)
[0189987e] -job initserver() = OK (0)
[0189987e] +job acceptconnections()
[0189987e] -job acceptconnections() = OK (0)
```

或者是在配置文件中配置，这样每次启动时将自动设置。

2. 命令操作

下面我们说明Swarm的常用命令，这里使用Swarm镜像来完成操作。在获得Swarm镜像之后，我们可以启动该镜像：

```
$ docker run --rm swarm
NAME:
    swarm - a Docker-native clustering system
USAGE:
    swarm [global options] command [command options] [arguments...]
VERSION:
    0.1.0 (a445ed9)
COMMANDS:
    create, c create a cluster
    list, l list nodes in a cluster
    manage, m manage a docker cluster
    join, j join a docker cluster
    help, h Shows a list of commands or help for one command


GLOBAL OPTIONS:
    --debug    debug mode [$DEBUG]
    --log-level, -l "info" Log level (options: debug, info, warn, error, fatal, panic)
    --help, -h    show help
    --version, -v  print the version
```

容器输出了Swarm的名字、用法、版本、命令和全局选项。可以看到，能够使用的命令有create、list、manage、join和help，其中help是帮助命令。下面我们分别说明这些命令的含义和用法。

创建集群

创建集群的命令为create，具体为：

```
$ docker run --rm swarm create
73f8bc512e941952l0fad6e9cd58986f
```

create命令会向Docker Hub的发现服务获得一个全球唯一的token，用于标识当前创建的Docker集群。这时Swarm集群并没有实际节点，只是创建了一个集群标识，Docker节点可以通过这个标识来加入到集群中成为Swarm节点。

加入集群

有了集群标识后，Docker节点就可以通过join命令加入到集群中：

```
$ docker run -d swarm join --addr=<node_ip:2375> token://<cluster_id>
```

其中node_ip就是Docker节点的IP，2375就是Docker后台监听的端口，cluster_id是刚刚通过create命令创建得到的集群id。例如，集群中有一个节点的地址为192.168.1.85，那么将其加入到刚建立的Swarm集群的操作为：

```
$ docker run -d swarm join --addr=192.168.1.85:2375 \ token://73f8bc512e941952l0fad6e9cd58986f
```

create命令是在Swarm主控节点上执行的，join命令是在需要加入到Swarm集群的Docker节点上执行的。Docker节点成为Swarm集群节点之后，Swarm主控节点就可以根据加入信息来发现该节点，从而获得该节点的状态信息，为调度模块决策提供支持。

管理集群

管理集群是通过manage命令来实现的，形如：

```
$ docker run -t -p <swarm_port>:2375 -t swarm manage token://<cluster_id>
```

例如，

```
$ docker run -t -p 2376:2375 -t swarm manage token:// 73f8bc512e941952l0fad6e9cd58986f
```

这条命令可以在Swarm集群的任何节点上运行，-p用于完成端口映射。manage命令执行之后，在Swarm集群内部，将会启动Swarm并接受后续的Docker集群管理请求。启动Swarm之后，我们就可以向Swarm服务器发送类似于Docker后台的请求，例如：

```
# use the regular docker cli
$ docker -H tcp://<swarm_ip:swarm_port> info
$ docker -H tcp://<swarm_ip:swarm_port> run ...
$ docker -H tcp://<swarm_ip:swarm_port> ps
$ docker -H tcp://<swarm_ip:swarm_port> logs ...
...
```

通过上面的命令，我们可以在该Swarm服务器上执行信息查询、执行命令、查看当前运行命令以及输出日志等。

列出集群节点

通过list命令可以列出集群中的节点状况，形如：

```
$ docker run --rm swarm list token://<cluster_id>
```

例如：

```
$ docker run --rm swarm list token:// 73f8bc512e94195210fad6e9cd58986f
192.168.1.85:2375
192.168.1.86:2375
```

18.2.3 发现服务和调度

由于发现服务和调度是Swarm的核心主体，所以在这一节中，我们将详细说明其原理和实际配置。

1. 发现服务

在18.2.1节中，我们讲到发现服务是一种维护集群状态的机制，它可以和各种后端服务器协调合作，例如Consul、Etcd以及直接使用节点发现和文件发现，但是不管采用什么方式，所有的方式都是为了维护Docker节点列表以及跟踪每个节点的健康状态和进出状态。

下面我们分别介绍节点发现、文件发现、Consul发现和Etcd发现以及ZooKeeper发现。

节点发现

节点发现是Swarm最基本的发现方法，它不需要用到任何文件或者后端服务器，仅通过命令行就可以实现。节点发现的启动命令为：

```
swarm manage \
    --discovery dockerhost01:2375,dockerhost02:2375,dockerhost03:2375 \
    -H=0.0.0.0:2375
```

其中dockerhost01等是Docker节点的主机名或者IP。除了上述方式外，也可以采用下面的方式达到同样的效果。

首先，构建一个集群：

```
# create a cluster
$ swarm create
6856663cdefdec325839a4b7e1de38e8
```

然后运行join命令将需要加入到集群的节点通过IP:Port的形式加入到集群中，具体为：

```
$ swarm join --addr= dockerhost01:2375 token:// 6856663cdefdec325839a4b7e1de38e8
$ swarm join --addr= dockerhost02:2375 token:// 6856663cdefdec325839a4b7e1de38e8
$ swarm join --addr= dockerhost03:2375 token:// 6856663cdefdec325839a4b7e1de38e8
```

然后在任意Docker客户端运行如下命令即可：

```
# start the manager on any machine or your laptop
$ swarm manage -H tcp://<swarm_ip:swarm_port> token://<cluster_id>
```

文件发现

文件发现利用在本地文件系统放置的文件来配置发现服务，例如/etc/swarm/cluster_config。文件内部每行为IP:Port的形式，每一行代表一个Swarm节点，具体示例为：

```
swarm manage \
    --discovery file:///etc/swarm/cluster_config \
    -H=0.0.0.0:2375
```

其中文件内容为：

```
#/etc/swarm/cluster_config
dockerhost01:2375
dockerhost02:2375
dockerhost03:2375
```

Consul发现

Docker Swarm也支持Consul发现，它利用Consul的键值对保存机制来保存节点的IP:Port值，用以构建集群。在这种发现机制下，每个Docker节点的Swarm客户端在使用swarm join命令加入到集群时都要指向Consul的HTTP接口。Swarm客户端通过下面的命令启动：

```
swarm join \
    --discovery consul://consulhost01/swarm \
    # This can be an internal IP as long as the other
    # Docker hosts can reach it.
    --addr=10.100.199.200:2375
```

Swarm主控节点从Consul中获得集群信息，它通过下面的命令启动管理：

```
swarm manage \
    --discovery consul://consulhost01/swarm \
    -H=0.0.0.0:2375
```

Etcd发现

Etcd也是一个分布式键值对保存服务，和Consul发现机制基本一样。Docker节点在加入Swarm集群时需要指向Etcd接口，Etcd通过心跳来维护节点列表信息。若Docker节点想要加入到Etcd发现机制的Swarm集群，可以使用如下命令：

```
swarm join \
    --discovery etcd://etcdhost01/swarm \
    --addr=10.100.199.200:2375
```

主控节点通过下面的命令启动集群管理：

```
swarm manage \
    --discovery etcd://etcdhost01/swarm \
    -H=0.0.0.0:2375
```

ZooKeeper发现

Swarm也支持ZooKeeper发现。和其他键值对存储模型一样，ZooKeeper使用一个ZK集合来保存列表信息以及根据Docker后台的运行状态动态维护该键值对列表。Swarm的主控节点也连接到该集合，并使用该集合下/swarm目录下的信息维护主机列表清单，然后定期进行健康检查。

Swarm客户端通过下面的命令启动：

```
swarm join \
    # All hosts in the ensemble should be listed
    --discovery zk://zkhost01,zkhost02,zkhost03/swarm \
    --addr=10.100.199.200
```

主控节点通过下面的命令启动管理：

```
swarm manage \
    --discovery zk://zkhost01,zkhost02,zkhost03/swarm \
    -H 0.0.0.0:2375
```

2. 调度

调度是决定任务由哪个Swarm节点来执行的过程，它包含条件过滤和调度策略两部分内容。条件过滤主要是通过标签来实现的。在Docker后台启动的时候，我们可以给它赋予一系列标签，例如：

```
docker -d \
    --label storage=ssd \
    --label zone=external \
    --label tier=data \
    -H tcp://0.0.0.0:2375
```

上述Docker后台启动时，通过--label参数设置了3个标签，并通过-H参数将Docker后台绑定到主机的2375端口。当节点启动后，Swarm就可以根据这些标签过滤选择节点。Swarm的过滤机制主要有以下几种。

约束过滤

约束过滤（Constraint Filter）是使用Docker后台启动时配置的键值对标签来过滤的。当用户需要创建一个容器时，他可以通过选取一组或者多组标签约束来选取符合条件的Swarm节点来创建和运行容器。这种约束主要的应用场景如下所示。

- 通过特定的主机特性进行选择，包括选取特定硬件的主机执行任务，例如storage=ssd。
- 选取特定位置的主机，例如选取美国东部的主机：region=us-east。当然，这个键值对必须在启动Docker后台时通过--label参数设定过了。
- 将集群进行逻辑切分，例如environment=production将集群配置为开发和产品两个子集群，然后选取其中的子集群运行任务。

下面给出一个简单的例子，例如启动node-1节点上的Docker后台，通过--label配置storage=ssd

标签：

```
$ docker -d --label storage=ssd
$ swarm join --addr=192.168.0.42:2375 token://XXXXXXXXXXXXXXXXX
```

同样，在node-2节点上启动Docker后台，配置storage=disk标签：

```
$ docker -d --label storage=disk
$ swarm join --addr=192.168.0.43:2375 token://XXXXXXXXXXXXXXXXX
```

两个节点都注册进了同一个集群，接着我们运行一个mysql服务容器，选择SSD存储方式的节点作为运行节点，具体操作为：

```
$ docker run -d -P -e constraint:storage==ssd --name db mysql
f8b693db9cd6
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NODE                NAMES
f8b693db9cd6        mysql:latest        "mysqld"            Less than a second ago   running
192.168.0.42:49178->3306/tcp    node-1      db
```

此时，node-1节点被选取。

接着，我们要在硬盘存储方式的节点上部署一个nginx服务：

```
$ docker run -d -P -e constraint:storage==disk --name frontend nginx
963841b138d8
$ docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED
STATUS              PORTS                          NODE                NAMES
963841b138d8        nginx:latest                   "nginx"             Less than a second ago
running             192.168.0.43:49177->80/tcp     node-2              frontend
f8b693db9cd6        mysql:latest                   "mysqld"            Up About a minute
running             192.168.0.42:49178->3306/tcp   node-1              db
```

可以看到，nginx服务容器部署到了node-2节点上。

除了上面所示的存储方式过滤外，经常拿来作为约束过滤条件的还有内核版本、操作系统、运行环境等。

类同过滤

类同过滤（Affinity Filter），是指参考另一个容器来部署新的容器，新的容器被部署到和被参考容器相同的节点上。

例如，我们把Nginx容器部署到了node-1节点上：

```
$ docker run -d -p 80:80 --name front nginx
87c4376856a8
$ docker ps
CONTAINER ID        IMAGE                COMMAND             CREATED
STATUS              PORTS                NODE                NAMES
87c4376856a8        nginx:latest         "nginx"             Less than a second ago
```

```
running             192.168.0.42:80->80/tcp        node-1              front
```

此时就可以通过-e affinity:container==front来启动另一个容器，使得该容器和front在同一个节点上：

```
$ docker run -d --name logger -e affinity:container==front logger
87c4376856a8
$ docker ps
CONTAINER ID        IMAGE                     COMMAND     CREATED
STATUS              PORTS                     NODE        NAMES
87c4376856a8        nginx:latest              "nginx"     Less than a second ago
running             192.168.0.42:80->80/tcp node-1      front
963841b138d8        logger:latest             "logger"    Less than a second ago
running
```

新启动的logger容器将会被部署到node-1节点上。

此外，可以通过类同过滤来避免从网络上拉取镜像，这样可以节省启动时间。类同过滤只将命令指派给那些本地就拥有该容器对应镜像的节点。

假如，我们已经运行了下面3条命令：

```
$ docker -H node-1:2375 pull redis
$ docker -H node-2:2375 pull mysql
$ docker -H node-3:2375 pull redis
```

此时就知道node-1和node-3已经拥有了redis镜像，那么当我们新启动redis容器时，可以使用-e affinity:image==redis来保证任务不会被调度到node-2，避免了拉取redis镜像带来的时耗，具体操作如下：

```
$ docker run -d --name redis1 -e affinity:image==redis redis
$ docker run -d --name redis2 -e affinity:image==redis redis
$ docker run -d --name redis3 -e affinity:image==redis redis
$ docker run -d --name redis4 -e affinity:image==redis redis
$ docker run -d --name redis5 -e affinity:image==redis redis
$ docker run -d --name redis6 -e affinity:image==redis redis
$ docker run -d --name redis7 -e affinity:image==redis redis
$ docker run -d --name redis8 -e affinity:image==redis redis
$ docker ps
CONTAINER ID        IMAGE              COMMAND             CREATED
TATUS               PORTS              NODE                NAMES
87c4376856a8        redis:latest       "redis"             Less than a second ago
running                                node-1              redis1
1212386856a8        redis:latest       "redis"             Less than a second ago
running                                node-1              redis2
87c4376639a8        redis:latest       "redis"             Less than a second ago
running                                node-3              redis3
1234376856a8        redis:latest       "redis"             Less than a second ago
running                                node-1              redis4
86c2136253a8        redis:latest       "redis"             Less than a second ago
running                                node-3              redis5
87c3236856a8        redis:latest       "redis"             Less than a second ago
```

```
running                                    node-3              redis6
87c4376856a8        redis:latest           "redis"             Less than a second ago
running                                    node-3              redis7
963841b138d8        redis:latest           "redis"             Less than a second ago
running
```

端口过滤

我们知道如果主机上的端口被一个容器的开放端口绑定后，就不能再被其他容器绑定了。Swarm中的端口过滤（Port Filter）指的就是当你指定端口的静态映射关系后，Swarm的调度器会选择尚能够满足你需求的节点来部署容器，如果整个集群都没有满足条件的节点，请求就会失败。

下面我们通过例子说明一下。我们的集群总共有3个节点：node-1、node-2和node-3。首先，我们绑定主机的80端口到nginx容器的80开放端口：

```
$ docker run -d -p 80:80 nginx
87c4376856a8
$ docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED
STATUS              PORTS                  NODE                NAMES
87c4376856a8        nginx:latest           "nginx"             Less than a second ago
running             192.168.0.42:80->80/tcp node-1             prickly_engelbart
```

可以看到，容器部署到了node-1节点上。接着，我们继续重复这个过程：

```
$ docker run -d -p 80:80 nginx
963841b138d8
$ docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED
STATUS              PORTS                  NODE                NAMES
963841b138d8        nginx:latest           "nginx"             Less than a second ago
running             192.168.0.43:80->80/tcp node-2             dreamy_turing
87c4376856a8        nginx:latest           "nginx"             Up About a minute
running             192.168.0.42:80->80/tcp node-1             prickly_engelbart
```

新的容器被部署到了node-2节点。

继续部署：

```
$ docker run -d -p 80:80 nginx
963841b138d8
$ docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED
STATUS              PORTS                  NODE                NAMES
f8b693db9cd6        nginx:latest           "nginx"             Less than a second ago
running             192.168.0.44:80->80/tcp node-3             stoic_albattani
963841b138d8        nginx:latest           "nginx"             Up About a minute
running             192.168.0.43:80->80/tcp node-2             dreamy_turing
87c4376856a8        nginx:latest           "nginx"             Up About a minute
running             192.168.0.42:80->80/tcp node-1             prickly_engelbart
```

第三个容器被部署到了node-3节点。继续部署：

```
$ docker run -d -p 80:80 nginx
2014/10/29 00:33:20 Error response from daemon: no resources available to schedule container
```

产生错误了。因为集群总共就有3个节点，而每次部署都静态绑定80端口，当3个节点的80端口被占用之后，调度就会失败，返回错误。

除了约束过滤、类同过滤和端口过滤外，还有依赖过滤和健康状况过滤等，在此不再展开。

下面简要说明调度的策略，主要包含Binpacking和随机调度，其中随机调度就是一个rand()函数，这里不再赘述。

Binpacking策略

Binpacking策略是尽可能紧凑地使用节点资源。它会根据节点的CPU和RAM等资源为节点排序，并挑选出能够让节点资源不产生空隙、碎片的节点来运行容器，为尚未有任务的节点保留最大的空间。

下面我们通过例子说明该策略。假设集群有两个节点node-1和node-2，它们都具有2GB的RAM存储，我们先部署一个mysql容器，指定占用1GB的内存：

```
$ docker run -d -P -m 1G --name db mysql
f8b693db9cd6
$ docker ps
CONTAINER ID        IMAGE                       COMMAND             CREATED
STATUS              PORTS                       NODE                NAMES
f8b693db9cd6        mysql:latest                "mysqld"            Less than a second ago
running             192.168.0.42:49178->3306/tcp node-1             db
```

可以看到，节点被部署到node-1，接着我们继续部署一个nginx容器，也占用1GB内存：

```
$ docker run -d -P -m 1G --name frontend nginx
963841b138d8
$ docker ps
CONTAINER ID        IMAGE                       COMMAND             CREATED
STATUS              PORTS                       NODE                NAMES
963841b138d8        nginx:latest                "nginx"             Less than a second ago
running             192.168.0.42:49177->80/tcp  node-1              frontend
f8b693db9cd6        mysql:latest                "mysqld"            Up About a minute
running             192.168.0.42:49178->3306/tcp node-1             db
```

结果显示，它也被部署到了node-1，为什么呢？因为在Binpacking策略下，nginx容器可以在node-1中部署，那么就尽量保留node-2，以备后续容器使用。

18.3 Compose

诸如前两节所说，Machine用于安装Docker，Swarm用于管理Docker集群，而本节的Compose则是管理Docker容器的应用部署。它所做的工作和Fig类似，都是安排应用部署到哪组容器以及容器之间的关系。事实上，Compose就是由Fig启发而来的，它的设计理念和配置文件都与Fig类

似。Compose也是采用YAML文件作为配置文件，只要编写好配置文件，一条命令就可以让多个相关容器跑起来。

下面我们来看看配置文件group.yml：

```
name: rails_example
containers:
    db:
        image: postgres:latest
    web:
        build: .
        command: bundle exec rackup -p 3000
        volumes:
            - .:/myapp

        ports:
            - "3000:3000"

        links:
            - db
```

我们在第15章中见过这个格式，其中name定义了该容器组的名字为rails_example。容器组包含两个容器：db和web。这是一个典型的Web应用程序，web容器根据当前目录下的Dockerfile文件构建，将当前目录映射为容器内的/myapp数据卷，将本地的3000端口和容器内的3000端口进行映射，连接到db容器。定义好配置及相关文件后，只需要执行docker up命令就可以将上面的容器组跑起来。需要注意的是，目前Docker里面并没有包含Compose组件，所以需要读者自行到https://github.com/docker/docker/issues/9459下载平台对应的版本，然后替换掉原有的Docker文件，具体操作如下：

```
root@micall-ThinkPad:~# curl -LO http://cl.ly/0Q3G1l2t301S/download/docker-1.3.2-dev-linux
root@micall-ThinkPad:~# /etc/init.d/docker stop
root@micall-ThinkPad:~# mv /usr/local/bin/docker ./docker-stable
root@micall-ThinkPad:~# mv ./docker-1.3.2-dev-linux /usr/local/bin/docker
root@micall-ThinkPad:~# chmod +x /usr/local/bin/docker
root@micall-ThinkPad:~# /etc/init.d/docker start
root@micall-ThinkPad:~# docker version
```

这样就替换掉了原有的Docker，置换成包含Compose组件的Docker。关于Compose的更多用法和内容，可以访问上述地址，但目前能够看到的资料不多。

Part 4 第四篇

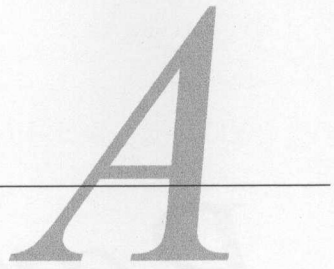
附　录

本篇内容

- 附录 A　常见镜像
- 附录 B　Docker API 列表
- 附录 C　参考资料

附录 A

常见镜像



本章将介绍一些常用的镜像，主要包含系统镜像、数据库镜像、Web服务镜像以及语言镜像。

A.1 系统镜像

系统镜像主要是提供系统环境或者基本工具集的镜像，例如BusyBox、Ubuntu以及CentOS等。

A.1.1 BusyBox

BusyBox是一个包含一百多个常用Linux命令和工具的工具集，例如常见的ls、cat和echo，以及更为复杂的grep、find、mount和telnet。若要使用BusyBox，只需要在Docker Hub中搜索即可：

```
$ docker search busybox
NAME                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
busybox                           Busybox base image.                             126                  [OK]
progrium/busybox                                                                  33                   [OK]
jeanblanchard/busybox-java        Minimal Docker image with Java                  14                   [OK]
jeanblanchard/busybox-tomcat      Minimal Docker image with Apache Tomcat         9                    [OK]
radial/busyboxplus                Full-chain, Internet enabled,                   4                    [OK]
                                  busybox made...
sequenceiq/busybox                                                                1                    [OK]
```

选择最流行的官方镜像：

```
$ sudo docker pull busybox
```

启动BusyBox容器，并运行grep命令：

```
$ docker run -it busybox
/ # grep
BusyBox v1.22.1 (2014-05-22 23:22:11 UTC) multi-call binary.
Usage: grep [-HhnlLoqvsriwFE] [-m N] [-A/B/C N] PATTERN/-e PATTERN.../-f FILE [FILE]...
Search for PATTERN in FILEs (or stdin)
    -H Add 'filename:' prefix
    -h Do not add 'filename:' prefix
```

A.1.2 Ubuntu

Ubuntu是非常流行的Linux系统，可以直接在Docker Hub中搜索：

```
$ docker search ubuntu
NAME                              DESCRIPTION                                     STARS  OFFICIAL  AUTOMATED
ubuntu                            Official Ubuntu base image                      1233              [OK]
dockerfile/ubuntu                 Trusted automated Ubuntu                        41                [OK]
                                  (http://www.ubunt...
ansible/ubuntu14.04-ansible       Ubuntu 14.04 LTS with ansible                   35                [OK]
dockerfile/ubuntu-desktop         Trusted automated Ubuntu Desktop                20                [OK]
                                  (LXDE) (h...
ubuntu-upstart                    Upst
```

选择官方镜像并运行ubuntu容器：

```
$ docker run -ti ubuntu:14.04 /bin/bash
root@a74a37bd4bc0:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@a74a37bd4bc0:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 14.04.1 LTS
Release: 14.04
Codename: trusty
root@a74a37bd4bc0:/#
```

A.1.3 CentOS

CentOS是Red Hat企业版源代码再编译的产物，常用于作为服务器，其性能稳定、可靠。在Docker Hub中可以直接使用其官方提供的镜像，例如使用`docker pull centos:6`和`docker pull centos:7`命令分别拉取CentOS 6和CentOS 7。

虽然CentOS 7已经将systemd服务替换为fakesystemd以解决依赖管理，但是有时我们还需要使用systemd组件，例如读取主机的cgroups等。下面简要说明如何集成systemd服务，具体的Dockerfile如下：

```
FROM centos:7
MAINTAINER "you" <your@email.here>
ENV container docker
RUN yum -y swap -- remove fakesystemd -- install systemd systemd-libs
RUN yum -y update; yum clean all; \
(cd /lib/systemd/system/sysinit.target.wants/; for i in *; do [ $i ==
systemd-tmpfiles-setup.service ] || rm -f $i; done); \
rm -f /lib/systemd/system/multi-user.target.wants/*;\
rm -f /etc/systemd/system/*.wants/*;\
rm -f /lib/systemd/system/local-fs.target.wants/*; \
rm -f /lib/systemd/system/sockets.target.wants/*udev*; \
rm -f /lib/systemd/system/sockets.target.wants/*initctl*; \
```

```
rm -f /lib/systemd/system/basic.target.wants/*;\
rm -f /lib/systemd/system/anaconda.target.wants/*;
VOLUME [ "/sys/fs/cgroup" ]
CMD ["/usr/sbin/init"]
```

该Dockerfile将fakesystemd删除，安装systemd服务，并删除一些可能引发冲突的文件。接着，可以使用该Dockerfile构建镜像：

```
docker build --rm -t local/c7-systemd .
```

下面我们在此基础上集成一个httpd服务，具体的Dockerfile为：

```
FROM local/c7-systemd
RUN yum -y install httpd; yum clean all; systemctl enable httpd.service
EXPOSE 80
CMD ["/usr/sbin/init"]
```

然后构建镜像并启动容器：

```
docker build --rm -t local/c7-systemd-httpd
docker run --privileged -ti -v /sys/fs/cgroup:/sys/fs/cgroup:ro -p 80:80 local/c7-systemd-httpd
```

至此，我们就可以通过浏览器访问本地的80端口，并且可以使用httpd提供的服务。

A.2　数据库镜像

数据库是信息系统及互联网程序必不可少的服务，用于存储软件使用或者产生的数据。目前，流行的数据库主要分为关系型数据库和非关系型数据库，前者以MySQL为代表，后者则有Redis、MongoDB和PostgreSQL等。

A.2.1　MySQL

MySQL是一款非常流行的开源关系型数据库，具有体积小、速度快、开源等优点。关系型数据库将数据保存在不同的表中，通过关系表将现实世界的关系进行建模。MySQL所使用的SQL语言是访问数据库的最常用的标准化语言。

运行如下命令，即可启动官方的mysql容器：

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
```

如果其他应用需要使用它，则可以通过如下命令连接到some-mysql容器：

```
$ docker run --name some-app --link some-mysql:mysql -d application-that-uses-mysql
```

在mysql容器中，有几个环境变量需要注意，具体如下所示。

- MYSQL_ROOT_PASSWORD：这是一个必须设置的环境变量，它设置的是MySQL的root用户的密码，上述例子中其值为mysecretpassword。

- MYSQL_USER和MYSQL_PASSWORD：这两个变量是可选变量，用于创建一个MySQL用户及其密码。需要注意的是，这两个变量必须同时设置，缺任何一个，另一个都将无效。使用这两个变量创建的用户将拥有MYSQL_DATABASE变量指定的数据库的所有权限。root用户无需通过该变量设置。
- MYSQL_DATABASE：可选变量，用于指定创建数据库的名字。如果MYSQL_USER和MYSQL_PASSWORD被设置，则该用户拥有该数据库的所有权限。

除了直接使用官方的镜像外，还可以通过Dockerfile来创建。Dockerfile的内容为：

```
FROM debian:wheezy
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever
    dependencies get added
RUN groupadd -r mysql && useradd -r -g mysql mysql
# FATAL ERROR: please install the following Perl modules before executing
/usr/local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
RUN apt-get update && apt-get install -y perl --no-install-recommends && rm -rf /var/lib/apt/lists/*
# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.oracle.com>" imported
RUN apt-key adv --keyserver pool.sks-keyservers.net --recv-keys
A4A9406876FCBD3C456770C88C718D3B5072E1F5
ENV MYSQL_MAJOR 5.7
ENV MYSQL_VERSION 5.7.5-m15
RUN echo "deb http://repo.mysql.com/apt/debian/ wheezy mysql-${MYSQL_MAJOR}-dmr" >
/etc/apt/sources.list.d/mysql.list
# the "/var/lib/mysql" stuff here is because the mysql-server postinst doesn't have an explicit way
    to disable the mysql_install_db codepath besides having a database already "configured" (ie, stuff in
/var/lib/mysql/mysql)
# also, we set debconf keys to make APT a little quieter
RUN { \
    echo mysql-community-server mysql-community-server/data-dir select ''; \
    echo mysql-community-server mysql-community-server/root-pass password ''; \
    echo mysql-community-server mysql-community-server/re-root-pass password ''; \
    echo mysql-community-server mysql-community-server/remove-test-db select false; \
 } | debconf-set-selections \
 && apt-get update && apt-get install -y mysql-server="${MYSQL_VERSION}"* && rm -rf /var/lib/apt/lists/* \
 && rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql
# comment out a few problematic configuration values
RUN sed -Ei 's/^(bind-address|log)/#&/' /etc/mysql/my.cnf
VOLUME /var/lib/mysql
COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 3306
CMD ["mysqld"]
```

保存好之后，使用docker build命令来创建镜像：

```
$ sudo docker build -t mysql:latest
```

至此，mysql镜像构建完毕。

A.2.2　Redis

Redis（全称是REmote DIctionary Server，远程字典服务器），是一款开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型的键值数据库。为了提高存储效率，Redis将数据存储在内存中，然后周期性地将数据更新到磁盘上，并在此基础上实现主从同步。

启动Redis容器：

```
$ docker run --name some-redis -d redis
```

镜像默认是对外暴露6379端口。也可以启动持久化存储：

```
$ docker run --name some-redis -d redis redis-server --appendonly yes
```

数据将会被放到VOLUME /data下，可以使用--volume-from连接数据容器或者使用-v、docker/host/dir:/data进行映射。

其他容器可以通过如下命令连接到some-redis容器：

```
$ docker run --name some-app --link some-redis:redis -d application-that-uses-redis
```

或者是：

```
$ docker run -it --link some-redis:redis --rm redis sh -c 'exec redis-cli -h "$REDIS_PORT_6379_TCP_ADDR"
    -p "$REDIS_PORT_6379_TCP_PORT"'
```

如果你想用自己的redis.conf配置文件，则可以通过如下命令：

```
docker run -v /myredis/conf/redis.conf:/usr/local/etc/redis/redis.conf --name myredis redis
    /usr/local/etc/redis/redis.conf
```

其中/myredis/conf/redis.conf是本地机器上的配置，它需要映射到容器内部。

当然，除了这种方法外，还可以使用Dockerfile构建一个新的Redis容器，其中Dockerfile的内容为：

```
FROM redis
COPY redis.conf /usr/local/etc/redis/redis.conf
CMD [ "redis-server", "/usr/local/etc/redis/redis.conf" ]
```

通过Dockerfile也可以构建和官方一样的Redis镜像。这里以2.8版本为例，其Dockerfile为：

```
FROM debian:wheezy
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever
dependencies get added
RUN groupadd -r redis && useradd -r -g redis redis
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
# grab gosu for easy step-down from root
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF357DD4
```

```
RUN curl -o /usr/local/bin/gosu -SL
    "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg --print-architecture)" \
    && curl -o /usr/local/bin/gosu.asc -SL
    "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg --print-architecture).asc" \
    && gpg --verify /usr/local/bin/gosu.asc \
    && rm /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu
ENV REDIS_VERSION 2.8.19
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-2.8.19.tar.gz
ENV REDIS_DOWNLOAD_SHA1 3e362f4770ac2fdbdce58a5aa951c1967e0facc8
# for redis-sentinel see: http://redis.io/topics/sentinel
RUN buildDeps='gcc libc6-dev make'; \
    set -x \
    && apt-get update && apt-get install -y $buildDeps --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/redis \
    && curl -sSL "$REDIS_DOWNLOAD_URL" -o redis.tar.gz \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | sha1sum -c - \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data
COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 6379
CMD [ "redis-server" ]
```

A.2.3 MongoDB

MongoDB是目前非常流行的非关系型数据库（NoSQL），它的每一条记录都是一个Document对象，非常适合大数据量、高并发、弱事务的互联网应用。特别是随着Web 2.0网站的兴起，MongoDB不仅可以满足移动互联网的数据存储需求，其开箱即用的特性也大大降低了中小型网站的运维成本，受到开发者的一致热捧。

要使用MongoDB，可以使用以下命令在后台启动：

```
$ docker run -d -p 27017:27017 --name mongodb dockerfile/mongodb
```

或者将其数据持久化到本地：

```
$ docker run -d -p 27017:27017 -v <db-dir>:/data/db --name mongodb dockerfile/mongodb
```

使用Dockerfile构建MongoDB镜像，其中Dockerfile的内容为：

```
FROM debian:wheezy
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever
```

```
dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb
RUN apt-get update \
    && apt-get install -y curl numactl \
    && rm -rf /var/lib/apt/lists/*
# grab gosu for easy step-down from root
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF357DD4
RUN curl -o /usr/local/bin/gosu -SL "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg
    --print-architecture)" \
    && curl -o /usr/local/bin/gosu.asc -SL
    "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg --print-architecture).asc" \
    && gpg --verify /usr/local/bin/gosu.asc \
    && rm /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu
ENV MONGO_RELEASE_FINGERPRINT BDC0DB28022D7DEA1490DC3E7085801C857FD301
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys $MONGO_RELEASE_FINGERPRINT
ENV MONGO_VERSION 3.0.0-rc6
RUN curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_VERSION.tgz" -o mongo.tgz \
    && curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_VERSION.tgz.sig" -o
    mongo.tgz.sig \
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*
VOLUME /data/db
COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 27017
CMD ["mongod"]
```

A.2.4　PostgreSQL

PostgreSQL，也常简称为Postgres，是一种对象-关系型数据库系统（ORDBMS）。它不仅支持关系型数据库的各种功能，而且具备类、继承等对象数据库的特征，是目前功能最全、特性最丰富的结构复杂的开源数据库系统，其本身的发展历程也见证了数据库理论和技术的发展历程。PostgreSQL之初是作为教学数据库推出的，有很多先进的功能，但易用性不如MySQL。不过随着8.x版本的推出，PostgreSQL越来越受到重视。

通过下面的命令可以启动一个postgre容器：

```
$ sudo docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

该容器默认对外开放5432端口。环境变量POSTGRES_PASSWORD用于为数据库的超级用户postgres设置密码。其他应用可以通过如下方式连接到该容器：

```
$ sudo docker run --name some-app --link some-postgres:postgres -d application-that-uses-postgres
```

或者是：

```
$ sudo docker run -it --link some-postgres:postgres --rm postgres sh -c 'exec psql -h
"$POSTGRES_PORT_5432_TCP_ADDR" -p "$POSTGRES_PORT_5432_TCP_PORT" -U postgres'
```

下面是构建9.4版本的postgresql镜像的Dockerfile：

```
# vim:set ft=dockerfile:
FROM debian:wheezy
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever
dependencies get added
RUN groupadd -r postgres && useradd -r -g postgres postgres
# grab gosu for easy step-down from root
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF357DD4
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/* \
    && curl -o /usr/local/bin/gosu -SL
    "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg --print-architecture)" \
    && curl -o /usr/local/bin/gosu.asc -SL
    "https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg --print-architecture).asc" \
    && gpg --verify /usr/local/bin/gosu.asc \
    && rm /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && apt-get purge -y --auto-remove curl
# make the "en_US.UTF-8" locale so postgres will be utf-8 enabled by default
RUN apt-get update && apt-get install -y locales && rm -rf /var/lib/apt/lists/* \
    && localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias en_US.UTF-8
ENV LANG en_US.utf8
RUN mkdir /docker-entrypoint-initdb.d
RUN apt-key adv --keyserver pool.sks-keyservers.net --recv-keys
B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
ENV PG_MAJOR 9.4
ENV PG_VERSION 9.4.0-1.pgdg70+1
RUN echo 'deb http://apt.postgresql.org/pub/repos/apt/ wheezy-pgdg main' $PG_MAJOR >
/etc/apt/sources.list.d/pgdg.list
RUN apt-get update \
    && apt-get install -y postgresql-common \
    && sed -ri 's/#(create_main_cluster) .*$/\1 = false/' /etc/postgresql-common/createcluster.conf \
    && apt-get install -y \
       postgresql-$PG_MAJOR=$PG_VERSION \
       postgresql-contrib-$PG_MAJOR=$PG_VERSION \
    && rm -rf /var/lib/apt/lists/*
RUN mkdir -p /var/run/postgresql && chown -R postgres /var/run/postgresql
ENV PATH /usr/lib/postgresql/$PG_MAJOR/bin:$PATH
ENV PGDATA /var/lib/postgresql/data
VOLUME /var/lib/postgresql/data
COPY docker-entrypoint.sh /

ENTRYPOINT ["/docker-entrypoint.sh"]
EXPOSE 5432
CMD ["postgres"]
```

使用Dockerfile构建镜像并启动容器：

```
$ sudo docker build -t eg_postgresql .
$ sudo docker run --rm -P --name pg_test eg_postgresql
```

有两种方法可以连接到PostgreSQL服务器。我们可以使用链接容器，或者从我们的主机（或网络）访问它。在客户端docker run命令中，直接使用--link remote_name:local_alias使容器连

接到另一个容器端口：

```
$ sudo docker run --rm -t -i --link pg_test:pg eg_postgresql bash
postgres@7ef98b1b7243:/$ psql -h $PG_PORT_5432_TCP_ADDR -p $PG_PORT_5432_TCP_PORT -d docker -U docker
-password
```

在上述命令中，我们通过psql客户端连接到PostgreSQL数据库。假设你有安装postgresql客户端，可以使用主机端口映射测试。你需要使用docker ps命令找出映射到本地主机的端口：

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND                CREATED
STATUS              PORTS                     NAMES
5e24362f27f6        eg_postgresql:latest      /usr/lib/postgresql/   About an hour ago
Up About an hour    0.0.0.0:49153->5432/tcp   pg_test

$ psql -h localhost -p 49153 -d docker -U docker --password
```

一旦你已经通过身份验证，并且有docker =#提示，就可以创建一个表并填充它：

```
psql (9.3.1)
Type "help" for help.
$ docker=# CREATE TABLE cities (
docker(#     name            varchar(80),
docker(#     location        point
docker(# );
CREATE TABLE
$ docker=# INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
INSERT 0 1
$ docker=# select * from cities;
     name      | location
---------------+-----------
 San Francisco | (-194,53)
(1 row)
```

此外，还可以使用busybox对其进行卷检查、定义日志文件、备份配置和数据等操作，相关代码为：

```
$ docker run --rm --volumes-from pg_test -t -i busybox sh
/ # ls
bin      etc      lib      linuxrc  mnt      proc     run      sys      usr
dev      home     lib64    media    opt      root     sbin     tmp      var
/ # ls /etc/postgresql/9.3/main/
environment      pg_hba.conf      postgresql.conf
pg_ctl.conf      pg_ident.conf    start.conf
/tmp # ls /var/log
ldconfig    postgresql
```

A.3　Web服务镜像

Web服务是互联网最主要的服务，它通过HTTP协议为客户端的请求提供响应。目前，常用

的Web服务有Apache httpd、Nginx和Tomcat。

A.3.1 httpd

httpd，即Apache HTTP服务器，也会直接简称为Apache。自1996年起，它就是世界流行度排名第一的Web服务器，可以运行在Windows、Linux等大多数操作系统之上。

通过以下命令可以直接启动httpd容器：

```
$ sudo docker run -it --rm --name my-apache-app -v "$(pwd)":/usr/local/apache2/htdocs/ httpd:2.4
```

此外，也可以使用Dockerfile的方式启动，其中Dockerfile的内容为：

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
```

然后通过如下命令构建镜像和启动容器：

```
docker build -t my-apache2 .
docker run -it --rm --name my-running-app my-apache2
```

除了直接使用官方的镜像外，读者还可以通过如下Dockerfile来构建一个httpd镜像：

```
FROM debian:jessie
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever
dependencies get added
#RUN groupadd -r www-data && useradd -r --create-home -g www-data www-data
ENV HTTPD_PREFIX /usr/local/apache2
ENV PATH $PATH:$HTTPD_PREFIX/bin
RUN mkdir -p "$HTTPD_PREFIX" \
    && chown www-data:www-data "$HTTPD_PREFIX"
WORKDIR $HTTPD_PREFIX
# install httpd runtime dependencies
# https://httpd.apache.org/docs/2.4/install.html#requirements
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
       libapr1 \
       libaprutil1 \
       libpcre++0 \
       libssl1.0.0 \
    && rm -r /var/lib/apt/lists/*
# see https://httpd.apache.org/download.cgi#verify
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys A93D62ECC3C8EA12DB220EC934EA76E6791485A8
ENV HTTPD_VERSION 2.4.12
ENV HTTPD_BZ2_URL https://www.apache.org/dist/httpd/httpd-$HTTPD_VERSION.tar.bz2
RUN buildDeps=' \
    ca-certificates \
    curl \
    bzip2 \
    gcc \
    libapr1-dev \
    libaprutil1-dev \
```

```
    libc6-dev \
    libpcre++-dev \
    libssl-dev \
    make \
    ' \
    set -x \
    && apt-get update \
    && apt-get install -y --no-install-recommends $buildDeps \
    && rm -r /var/lib/apt/lists/* \
    && curl -SL "$HTTPD_BZ2_URL" -o httpd.tar.bz2 \
    && curl -SL "$HTTPD_BZ2_URL.asc" -o httpd.tar.bz2.asc \
    && gpg --verify httpd.tar.bz2.asc \
    && mkdir -p src/httpd \
    && tar -xvf httpd.tar.bz2 -C src/httpd --strip-components=1 \
    && rm httpd.tar.bz2* \
    && cd src/httpd \
    && ./configure --enable-so --enable-ssl --prefix=$HTTPD_PREFIX \
    && make -j"$(nproc)" \
    && make install \
    && cd ../../ \
    && rm -r src/httpd \
    && sed -ri ' \
       s!^(\s*CustomLog)\s+\S+!\1 /proc/self/fd/1!g; \
       s!^(\s*ErrorLog)\s+\S+!\1 /proc/self/fd/2!g; \
       ' /usr/local/apache2/conf/httpd.conf \
    && apt-get purge -y --auto-remove $buildDeps
COPY httpd-foreground /usr/local/bin/
EXPOSE 80
CMD ["httpd-foreground"]
```

A.3.2　Nginx

Nginx是一个高性能的Web服务器，也可以作为HTTP、HTTPS、SMTP、TOP3、IMAP等应用的反向服务器。Nginx具有高并发、高性能和低内存耗费等优点，越来越受到Web应用开发者的欢迎。

通过如下命令可以启动Nginx容器作为一个静态页面的服务器：

```
$ sudo docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx
```

也可以使用Dockerfile来达到上述目的，其中Dockerfile的内容为：

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

然后使用docker build -t some-content-nginx命令构建镜像，然后启动该镜像：

```
docker run --name some-nginx -d some-content-nginx
```

暴露端口：

```
docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

可以通过浏览器来访问http://localhost:8080来验证服务是否正常工作。

Nginx的配置文件为nginx.conf，如果想自定义配置，则可以挂载本地的配置文件到容器内：

```
$ sudo docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx
```

同样，这个配置也可以通过Dockerfile的方式来实现：

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

使用docker build -t some-custom-nginx构建镜像，然后启动容器：

```
$ sudo docker run --name some-nginx -d some-custom-nginx
```

Nginx的官方仓库还提供了Dockerfile构建Nginx镜像，目前的版本是1.7.9，其内容为：

```
FROM debian:wheezy
MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"
RUN apt-key adv --keyserver pgp.mit.edu --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ wheezy nginx" >> /etc/apt/sources.list
ENV NGINX_VERSION 1.7.9-1~wheezy
RUN apt-get update && apt-get install -y nginx=${NGINX_VERSION} && rm -rf /var/lib/apt/lists/*
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log
VOLUME ["/var/cache/nginx"]
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

A.3.3 Tomcat

Tomcat是Apache软件基金会（Apache Software Foundation）开发的一款开源Web服务器和servlet容器，主要用在开发和部署Java servlet和JSP方面的应用上。

可以通过以下命令启动默认的服务容器：

```
$ sudo docker run -it --rm tomcat:8.0
```

或者指定端口映射：

```
$sudo docker run -it --rm -p 8888:8080 tomcat:8.0
```

这样通过浏览器浏览本地的8888端口，就可以访问其服务了。

Tomcat有几个环境变量非常重要。当用户需要自定义配置时，会用到这些环境变量，其默认值为：

```
CATALINA_BASE:   /usr/local/tomcat
```

```
CATALINA_HOME:   /usr/local/tomcat
CATALINA_TMPDIR: /usr/local/tomcat/temp
JRE_HOME:        /usr
CLASSPATH:       /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar
```

此外，Tomcat的配置文件在/usr/local/tomcat/conf/目录下。例如，你可以改变该目录下的tomcat-users.xml文件，加入用户使其可以成为manager-gui角色等。

下面我们给出Tomcat 8.0.18-jre8版本的Dockerfile：

```
FROM java:8-jre
ENV CATALINA_HOME /usr/local/tomcat
ENV PATH $CATALINA_HOME/bin:$PATH
RUN mkdir -p "$CATALINA_HOME"
WORKDIR $CATALINA_HOME
# see https://www.apache.org/dist/tomcat/tomcat-8/KEYS
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys \
    05AB33110949707C93A279E3D3EFE6B686867BA6 \
    07E48665A34DCAFAE522E5E6266191C37C037D42 \
    47309207D818FFD8DCD3F83F1931D684307A10A5 \
    541FBE7D8F78B25E055DDEE13C370389288584E7 \
    61B832AC2F1C5A90F0F9B00A1C506407564C17A3 \
    79F7026C690BAA50B92CD8B66A3AD3F4F22C4FED \
    9BA44C2621385CB966EBA586F72C284D731FABEE \
    A27677289986DB50844682F8ACB77FC2E86E29AC \
    A9C5DF4D22E99998D9875A5110C01C5A2F6059E7 \
    DCFD35E0BF8CA7344752DE8B6FB21E8933C60243 \
    F3A04C595DB5B6A5F1ECA43E3B7BBB100D811BBE \
    F7DA48BB64BCB84ECBA7EE6935CD23C10D498E23
ENV TOMCAT_MAJOR 8
ENV TOMCAT_VERSION 8.0.18
ENV TOMCAT_TGZ_URL
https://www.apache.org/dist/tomcat/tomcat-$TOMCAT_MAJOR/v$TOMCAT_VERSION/bin/apache-tomcat-$TOMCAT
    _VERSION.tar.gz
RUN curl -SL "$TOMCAT_TGZ_URL" -o tomcat.tar.gz \
    && curl -SL "$TOMCAT_TGZ_URL.asc" -o tomcat.tar.gz.asc \
    && gpg --verify tomcat.tar.gz.asc \
    && tar -xvf tomcat.tar.gz --strip-components=1 \
    && rm bin/*.bat \
    && rm tomcat.tar.gz*
EXPOSE 8080
CMD ["catalina.sh", "run"]
GitLib 镜像
```

A.4　语言镜像

语言镜像主要是说明各种编程语言的开发环境镜像。

A.4.1 Python

Python是一门交互式的、面向对象的解释型计算机编程语言。它支持模块化、异常处理、动态类型以及类等特性。Python常常被称作胶水语言，因为它能够把其他语言制作的各种模块很轻松地连接在一起，特别是C/C++实现的模块。由于Python语言简洁、易读和可扩展，Python已经成为了一种流行语言。常见的情形就是使用Python开发应用原型，然后使用其他语言实现其中的子模块。Python可以运行在Linux、iOS X以及Windows等系统上。目前，最流行的版本是2.7.X版本，也出了3.X版本，前者成为Python 2，后者称为Python 3。Python 3改动较大，很多地方并不兼容Python 2，这点需要特别注意。

下面的命令将使用Python 2来运行一个Python脚本：

```
$sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp python:2
python your-daemon-or-script.py
```

对应的Python 3版本的代码为：

```
$sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp python:3
python your-daemon-or-script.py
```

此外，也可以在镜像的基础上创建一个简单的Dockerfile来实现这一目标。下面以Python 2为例进行介绍：

```
FROM python:2-onbuild
CMD [ "python", "./your-daemon-or-script.py" ]
```

然后执行：

```
docker build -t my-python-app .
docker run -it --rm --name my-running-app my-python-app
```

当然，也可以使用Dockerfile文件来构建Python镜像。这里以Python 2.7为例，下面是Dockerfile的内容：

```
FROM buildpack-deps:wheezy
# remove several traces of debian python
RUN apt-get purge -y python.*
# http://bugs.python.org/issue19846
# > At the moment, setting "LANG=C" on a Linux system *fundamentally breaks Python 3*, and that's not
    OK.
ENV LANG C.UTF-8
ENV PYTHON_VERSION 2.7.8
RUN set -x \
    && mkdir -p /usr/src/python \
    && curl -SL "https://www.python.org/ftp/python/$PYTHON_VERSION/Python-$PYTHON_VERSION.tar.xz" \
       | tar -xJC /usr/src/python --strip-components=1 \
    && cd /usr/src/python \
    && ./configure --enable-shared \
    && make -j$(nproc) \
```

```
    && make install \
    && ldconfig \
    && curl -SL 'https://bootstrap.pypa.io/get-pip.py' | python2 \
    && find /usr/local \
       \( -type d -a -name test -o -name tests \) \
       -o \( -type f -a -name '*.pyc' -o -name '*.pyo' \) \
       -exec rm -rf '{}' + \
    && rm -rf /usr/src/python
# install "virtualenv", since the vast majority of users of this image will want it
RUN pip install virtualenv
CMD ["python2"]
```

A.4.2　Java

Java是一种跨平台的面向对象的程序设计语言，具有很好的通用性、高效性、移植性和安全性，广泛应用于个人电脑、数据中心、游戏控制台、移动设备等平台，其开发社区极为活跃，拥有广泛全面的程序库供开发者引用开发。

下面的Dockerfile可以将Java镜像作为编译和运行环境，不仅编译Main.java源文件为Main.class，而且在容器启动时会在容器内部执行Main：

```
FROM java:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

构建镜像并启动容器：

```
docker build -t my-java-app .
docker run -it --rm --name my-running-app my-java-app
```

如果仅仅想编译源代码，则可以直接通过命令行，具体为：

```
docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp java:7 javac Main.java
```

下面给出通过Dockerfile构建Java镜像的Dockerfile（这里以openjdk-8为例）：

```
FROM buildpack-deps:sid-scm
# A few problems with compiling Java from source:
#  1. Oracle.  Licensing prevents us from redistributing the official JDK.
#  2. Compiling OpenJDK also requires the JDK to be installed, and it gets
#       really hairy.
RUN apt-get update && apt-get install -y unzip && rm -rf /var/lib/apt/lists/*
ENV JAVA_VERSION 8u40~b22
ENV JAVA_DEBIAN_VERSION 8u40~b22-2
# see https://bugs.debian.org/775775
# and https://github.com/docker-library/java/issues/19#issuecomment-70546872
ENV CA_CERTIFICATES_JAVA_VERSION 20140324
RUN apt-get update && apt-get install -y openjdk-8-jdk="$JAVA_DEBIAN_VERSION"
ca-certificates-java="$CA_CERTIFICATES_JAVA_VERSION" && rm -rf /var/lib/apt/lists/*
```

```
# see CA_CERTIFICATES_JAVA_VERSION notes above
RUN /var/lib/dpkg/info/ca-certificates-java.postinst configure
# If you're reading this and have any feedback on how this image could be
#   improved, please open an issue or a pull request so we can discuss it!
```

openjdk-8-jre的Dockerfile内容为：

```
FROM buildpack-deps:sid-curl
# A few problems with compiling Java from source:
#  1. Oracle.  Licensing prevents us from redistributing the official JDK.
#  2. Compiling OpenJDK also requires the JDK to be installed, and it gets
#       really hairy.
RUN apt-get update && apt-get install -y unzip && rm -rf /var/lib/apt/lists/*
ENV JAVA_VERSION 8u40~b22
ENV JAVA_DEBIAN_VERSION 8u40~b22-2
# see https://bugs.debian.org/775775
# and https://github.com/docker-library/java/issues/19#issuecomment-70546872
ENV CA_CERTIFICATES_JAVA_VERSION 20140324
RUN apt-get update && apt-get install -y openjdk-8-jre-headless="$JAVA_DEBIAN_VERSION"
ca-certificates-java="$CA_CERTIFICATES_JAVA_VERSION" && rm -rf /var/lib/apt/lists/*
# see CA_CERTIFICATES_JAVA_VERSION notes above
RUN /var/lib/dpkg/info/ca-certificates-java.postinst configure
# If you're reading this and have any feedback on how this image could be
#   improved, please open an issue or a pull request so we can discuss it!
```

A.4.3 Perl

Perl是一种解释型的动态编程语言，属于高级语言。它最初由Larry Wall设计，借鉴了C、sed、awk、shell脚本等语言，并在内部集成了正则表达式，以及巨大的第三方代码库CPAN。Perl一开始是作为Unix系统的管理小工具，后来演变为一门编程语言，用作Web编程、数据库处理等方面，所以它特别适合系统管理和Web编程。

通过下面命令即可使用Perl容器执行一个Perl脚本：

```
$sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp
    perl:5.20 perl your-daemon-or-script.pl
```

也可以使用Dockerfile来配置需要执行的Perl项目，其中Dockerfile的内容为：

```
FROM perl:5.20
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
CMD [ "perl", "./your-daemon-or-script.pl" ]
```

使用docker build命令构建镜像并启动容器：

```
docker build -t my-perl-app .
docker run -it --rm --name my-running-app my-perl-app
```

下面使用Dockerfile构建Perl镜像，其中Dockerfile的内容为：

```
FROM buildpack-deps
MAINTAINER Peter Martini <PeterCMartini@GMail.com>
RUN apt-get update \
    && apt-get install -y curl procps \
    && rm -fr /var/lib/apt/lists/*
RUN mkdir /usr/src/perl
WORKDIR /usr/src/perl
COPY sha1.txt /tmp/sha1.txt
RUN curl -SL https://cpan.metacpan.org/authors/id/R/RJ/RJBS/perl-5.18.4.tar.bz2 -o
perl-5.18.4.tar.bz2 \
    && sha1sum -c /tmp/sha1.txt \
    && tar --strip-components=1 -xjf perl-5.18.4.tar.bz2 -C /usr/src/perl \
    && rm perl-5.18.4.tar.bz2 /tmp/sha1.txt \
    && ./Configure -Duse64bitall -A ccflags=-fwrapv -des \
    && make -j$(nproc) \
    && TEST_JOBS=$(nproc) make test_harness \
    && make install \
    && cd /usr/src \
    && curl -LO https://raw.githubusercontent.com/miyagawa/cpanminus/master/cpanm \
    && chmod +x cpanm \
    && ./cpanm App::cpanminus \
    && rm -fr ./cpanm /root/.cpanm /usr/src/perl
WORKDIR /root
CMD ["perl5.18.4","-de0"]
```

A.4.4　Ruby

Ruby是一门简单快捷的面向对象的服务器脚本语言，具有语法简单、扩展性强等特点，常用于编写CGI脚本、嵌入HTML中等。它可运行于多种平台，如Windows、Mac OS和UNIX的各种版本。

如果是运行单个文件的脚本，可以通过命令行的方式解释运行，具体操作为：

```
docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp ruby:2.1 ruby
your-daemon-or-script.rb
```

如果工程更为复杂，可以通过构建工程的Dockerfile，构建出和工程对应的镜像、容器。Dockerfile的内容为：

```
FROM ruby:2.1-onbuild
CMD ["./your-daemon-or-script.rb"]
```

然后将该Dockerfile文件复制到项目的根目录下，紧邻Gemfile文件。构建镜像时，将会执行COPY . /usr/src/app和RUN bundle install操作，前者将整个项目复制进镜像内，后者则是执行安装。

运行下面的命令构建镜像并启动容器：

```
docker build -t my-ruby-app .
docker run -it --name my-running-script my-ruby-app
```

下面是构建Ruby镜像的Dockerfile：

```
FROM buildpack-deps:wheezy
ENV RUBY_MAJOR 2.2
ENV RUBY_VERSION 2.2.0
# some of ruby's build scripts are written in ruby
# we purge this later to make sure our final image uses what we just built
RUN apt-get update \
    && apt-get install -y bison libgdbm-dev ruby \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/ruby \
    && curl -SL "http://cache.ruby-lang.org/pub/ruby/$RUBY_MAJOR/ruby-$RUBY_VERSION.tar.bz2" \
       | tar -xjC /usr/src/ruby --strip-components=1 \
    && cd /usr/src/ruby \
    && autoconf \
    && ./configure --disable-install-doc \
    && make -j"$(nproc)" \
    && make install \
    && apt-get purge -y --auto-remove bison libgdbm-dev ruby \
    && rm -r /usr/src/ruby
# skip installing gem documentation
RUN echo 'gem: --no-rdoc --no-ri' >> "$HOME/.gemrc"
# install things globally, for great justice
ENV GEM_HOME /usr/local/bundle
ENV PATH $GEM_HOME/bin:$PATH
RUN gem install bundler \
    && bundle config --global path "$GEM_HOME" \
    && bundle config --global bin "$GEM_HOME/bin"
# don't create ".bundle" in all our apps
ENV BUNDLE_APP_CONFIG $GEM_HOME
CMD [ "irb" ]
```

A.4.5 Node.js

Node.js是一个基于JavaScript运行时建立的网络平台，用于快速搭建易扩展的网络服务应用。Node.js采用非阻塞I/O和异步事件驱动模型，以达到最大化吞吐量和高效率，非常适合实时性要求高的场景。Node.js内部采用Google V8 JavaScript引擎执行代码，其大部分模块也是采用JavaScript编写。Node.js内部包含socket和HTTP的异步通信库，使得Node.js可以直接充当Web服务器，而不再需要额外的服务器。

如果是单文件的项目，则可以直接采用命令行的方式使用Node.js容器运行该项目，具体操作为：

```
docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp node:0.10
    node your-daemon-or-script.js
```

也可以在Node.js项目的根目录下创建一个Dockerfile，其内容为：

```
FROM node:0.10-onbuild
# replace this with your application's default port

EXPOSE 8888
```

然后构建镜像和启动容器：

```
docker build -t my-nodejs-app .
docker run -it --rm --name my-running-app my-nodejs-app
```

需要注意的是，项目必须包含package.json文件，用于列举出项目的依赖库。

也可以采用Dockerfile方式构建Node.js镜像，其Dockerfile的内容为：

```
FROM buildpack-deps:jessie
# verify gpg and sha256: http://nodejs.org/dist/v0.10.30/SHASUMS256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
# gpg: aka "Julien Gilli <jgilli@fastmail.fm>"
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D
114F43EE0176B71C7BC219DD50A3051F888C628D
ENV NODE_VERSION 0.8.28
ENV NPM_VERSION 2.4.1
RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc | sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/local --strip-components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.asc \
    && npm install -g npm@1.4.28 \
    && npm install -g npm@"$NPM_VERSION" \
    && npm cache clear
# note: we have to install npm 1.4.28 first because we can't go straight from 1.2 -> 2.0
# see also https://github.com/docker-library/node/issues/15#issuecomment-57879931
CMD [ "node" ]
```

A.4.6　Go

Go语言是Google公司在2009年推出的一款编程语言，主要针对多处理器系统应用程序的编程进行了优化，它不仅在执行速度上不比C/C++逊色，而且增加了垃圾回收、动态类型以及库管理等功能。Go语言常用于网络服务器的开发中，Google的App Engine以及Docker都是采用Go语言进行开发的。

使用Go容器最简单的方式是将它作为编译和运行的环境，此时只需要在项目的根目录下添加如下Dockerfile即可：

```
FROM golang:1.3-onbuild
```

虽然只有简单的一行代码，但通过该Dockerfile构建的镜像具有ONBUILD触发器，它将会执行COPY . /usr/src/app、RUN go get -d -v 、RUN go install -v和CMD["app"]等命令，分别对项目源码进行复制、编译、安装和运行。

通过下面的命令构建镜像和启动容器：

```
docker build -t my-golang-app .
docker run -it --rm --name my-running-app my-golang-app
```

如果你只是想通过它编译项目而不是在容器内部运行该项目，则可以通过下面的命令：

```
docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3 go build -v
```

该命令将当前目录映射到容器内的/usr/src/myapp，并将该目录作为工作目录，go build则仅编译项目，并将项目输出到/usr/sr/myapp中。

如果你的项目有Makefile，则可以通过下面的命令进行编译：

```
docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3 make
```

除了编译本地机器运行的程序外，还可以用Go进行交叉编译。例如，在Linux/amd64位机器上编译Windows/386目标的代码如下：

```
docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp -e GOOS=windows -e GOARCH=386
golang:1.3-cross go build -v
```

不仅如此，你还可以一次性编译出所有平台下的目标，具体为：

```
docker run --rm -it -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3-cross bash
$ for GOOS in darwin linux; do
>   for GOARCH in 386 amd64; do
>     go build -v -o myapp-$GOOS-$GOARCH
>   done
> done
```

最后，我们给出Go镜像的Dockerfile：

```
FROM buildpack-deps:jessie-scm
# gcc for cgo
RUN apt-get update && apt-get install -y \
    gcc libc6-dev make \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*
ENV GOLANG_VERSION 1.4.1
RUN curl -sSL https://golang.org/dl/go$GOLANG_VERSION.src.tar.gz \
    | tar -v -C /usr/src -xz
RUN cd /usr/src/go/src && ./make.bash --no-clean 2>&1
ENV PATH /usr/src/go/bin:$PATH
RUN mkdir -p /go/src
ENV GOPATH /go
ENV PATH /go/bin:$PATH
WORKDIR /go
COPY go-wrapper /usr/local/bin/
```

附录 B Docker API列表

为了更好地让读者查阅Docker的API，我们在这一章中按照容器相关和镜像相关两个部分来说明Docker的API。

B.1 容器相关的 API

容器相关的API主要是对容器的操作，包括列出容器、创建容器、查看容器、获取容器日志、启动和停止容器等操作。

1. 列出容器

方法： GET /containers/json

用例请求： GET /containers/json?all=1&before=8dfafdbc3a40&size=1 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    [
        {
            "Id": "8dfafdbc3a40",
            "Image": "base:latest",
            "Command": "echo 1",
            "Created": 1367854155,
            "Status": "Exit 0",
            "Ports":[{"PrivatePort": 2222, "PublicPort": 3333, "Type": "tcp"}],
            "SizeRw":12288,
            "SizeRootFs":0
        },
        ...
    ]
```

请求参数

- all ：其值为1/True/true或0/False/false，表示是否显示所有容器。其值若为真，则会显示所有容器，其中包含已经停止的容器，否则只显示正在运行的容器。其默认值为0。

- limit：仅显示最新建立的几个容器。
- since：显示比指定Id的容器更晚创建的容器。
- before：显示比指定Id的容器更早创建的容器。
- size：其值为1/True/true或者0/False/false，表示是否显示容器的大小。
- filters：使用JSON格式的条件过滤容器，例如退出码（exited=<int>），status运行状态（status=restarting|running|paused|exited）。

状态码

- 200：返回正常。
- 400：参数错误。
- 500：服务器错误。

curl操作：

```
curl -X GET http://localhost:2376/containers/json?all=1
```

2. 创建容器

方法：POST /containers/create

用例请求：

```
POST /containers/create HTTP/1.1
    Content-Type: application/json
    {
        "Hostname":"",
        "Domainname": "",
        "User":"",
        "Memory":0,
        "MemorySwap":0,
        "CpuShares": 512,
        "Cpuset": "0,1",
        "AttachStdin":false,
        "AttachStdout":true,
        "AttachStderr":true,
        "Tty":false,
        "OpenStdin":false,
        "StdinOnce":false,
        "Env":null,
        "Cmd":[
            "date"
        ],
        "Entrypoint": "",
        "Image":"base",
        "Volumes":{
            "/tmp": {}
        },
        "WorkingDir":"",
        "NetworkDisabled": false,
```

```
        "MacAddress":"12:34:56:78:9a:bc",
        "ExposedPorts":{
            "22/tcp": {}
        },
        "SecurityOpts": [""],
        "HostConfig": {
            "Binds":["/tmp:/tmp"],
            "Links":["redis3:redis"],
            "LxcConf":{"lxc.utsname":"docker"},
            "PortBindings":{ "22/tcp": [{ "HostPort": "11022" }] },
            "PublishAllPorts":false,
            "Privileged":false,
            "Dns": ["8.8.8.8"],
            "DnsSearch": [""],
            "VolumesFrom": ["parent", "other:ro"],
            "CapAdd": ["NET_ADMIN"],
            "CapDrop": ["MKNOD"],
            "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
            "NetworkMode": "bridge",
            "Devices": []
        }
    }
```

用例返回：

```
HTTP/1.1 201 Created
    Content-Type: application/json
    {
        "Id":"e90e34656806"
        "Warnings":[]
    }
```

JSON参数

- Hostname：容器内系统的主机名。
- Domainname：容器内系统的域名。
- User：容器内用户。
- Memory：内存（字节）。
- MemorySwap：包含Swap交换存储在内的总存储量。
- AttachStdin：是否附加到标准输入。
- AttachStdout：是否附加到标准输出。
- AttachStderr：是否附加到标准错误输出。
- Tty：是否需要附加伪终端。
- Env：环境变量，形如VAR=value。
- Cmd：容器要运行的命令。
- Entrypoint：容器的入口点。
- Image：容器基于的镜像。

- Volumes：数据卷。
- WorkingDir：默认工作目录。
- NetworkDisabled：是否打开网络。
- ExposedPorts：暴露端口，形如"ExposedPorts": { "<port>/<tcp|udp>: {}" }。
- SecurityOpts：安全选项，例如配置SELinux。
- HostConfig：子项配置。
- Binds：数据卷配置，形如host_path:container_path:ro。
- Links：容器连接，形如container_name:alias。
- PortBindings：端口映射，形如{ <port>/<protocol>: [{ "HostPort": "<port>" }] }。
- PublishAllPorts：是否给容器所有开放端口都随机映射到主机端口。
- Privileged：是否给予容器root权限访问宿主主机。
- Dns：Dns列表。
- DnsSearch：Dns搜索域。
- VolumesFrom：数据容器引用，形如<container name>[:<ro|rw>]。
- CapAdd：容器所使用的内核机制列表。
- Capdrop：容器不适用的内核机制列表。
- RestartPolicy：容器退出时的重启机制。
- NetworkMode：容器的网络类型，支持bridge、host和container:<name|id>。

请求参数

- name：容器的名字。

状态码

- 201：返回正常。
- 404：无该容器。
- 406：容器不能附加为终端（容器已经停止）。
- 500 ：服务器错误。

curl操作：

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:2376/containers/create -d '{
    "Hostname":"",
    ...
}'
```

3. 查看容器信息

方法：GET /containers/(id)/json

用例请求：GET /containers/4fa6e0f0c678/json HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {
        "Id": "4fa6e0f0c6786287e131c3852c58a2e01cc697a68231826813597e4994f1d6e2",
        "Created": "2013-05-07T14:51:42.041847+02:00",
        "Path": "date",
        "Args": [],
        ...
    }
```

参数：（无）

状态码

- 200：返回正常。
- 404：无该容器。
- 500：服务器错误。

curl操作：

```
curl -X GET http://localhost:2376/containers/4fa6e0f0c678/json
```

4. 查看容器正在运行的进程

方法：GET /containers/(id)/top

用例请求：GET /containers/4fa6e0f0c678/top HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {
        "Titles":[
            "USER",
            "PID",
            "%CPU",
            "%MEM",
            "VSZ",
            "RSS",
            "TTY",
            "STAT",
            "START",
            "TIME",
            "COMMAND"
        ],
        "Processes":[
            ["root","20147","0.0","0.1","18060","1864","pts/4","S","10:06","0:00","bash"],
            ["root","20271","0.0","0.0","4312","352","pts/4","S+","10:07","0:00","sleep","10"]
        ]
    }
```

参数

- ps_arg：ps命令的参数。

状态码：（略，重复状态码后续不再列出）

curl操作：

```
curl -X GET http://localhost:2376/containers/4fa6e0f0c678/top
```

5. 获取容器日志

获取指定容器的标准输出和标准错误输出。

方法：GET /containers/(id)/logs

用例请求：GET /containers/4fa6e0f0c678/logs?stderr=1&stdout=1&timestamps=1&follow=1&tail=10 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/vnd.docker.raw-stream
    {{ STREAM }}
```

参数

- follow：1/True/true或者0/False/false，流式返回，默认值为false。
- stdout：1/True/true或者0/False/false，显示标准输出，默认值为false。
- stderr：1/True/true或者0/False/false，显示标准错误输出，默认值为false。
- timestamps：1/True/true或者0/False/false，打印日志时间戳。
- tail：只打印末尾的N行。

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376/containers/4fa6e0f0c678/logs?stderr=1&stdout=1&timestamps
    =1&follow=1& tail=10
```

6. 查看容器的文件系统的变更

方法：GET /containers/(id)/changes

用例请求：GET /containers/4fa6e0f0c678/changes HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    [
        {
```

```
            "Path":"/dev",
            "Kind":0
        },
    ]
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376/containers/4fa6e0f0c678/changes
```

7. 导出容器

方法：GET /containers/(id)/export

用例请求：GET /containers/4fa6e0f0c678/export HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/octet-stream
    {{ TAR STREAM }}
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376/containers/4fa6e0f0c678/export
```

8. 启动容器

方法：POST /containers/(id)/start

用例请求：

```
POST /containers/(id)/start HTTP/1.1
   Content-Type: application/json
```

用例返回：HTTP/1.1 204 No Content

参数：（无）

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678/start
```

9. 停止容器

方法：POST /containers/(id)/stop

用例请求：POST /containers/e90e34656806/stop?t=5 HTTP/1.1

用例返回：HTTP/1.1 204 No Content

参数

- t：延时多少秒后停止。

状态码

- 304：容器已经停止。

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678/stop
```

10. 重启容器

方法：POST /containers/(id)/restart

用例请求：POST /containers/e90e34656806/restart?t=5 HTTP/1.1

用例返回：HTTP/1.1 204 No Content

参数

- t：延时多少秒后重启。

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678/restart
```

11. 杀死容器

方法：POST /containers/(id)/kill

用例请求：POST /containers/e90e34656806/kill HTTP/1.1

用例返回：HTTP/1.1 204 No Content

参数：

- signal：发送给容器的信号，如SIGINIT和SIGKILL。

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678/kill?signal=SIGKILL
```

12. 附加终端到容器

方法： POST /containers/(id)/attach

用例请求： POST /containers/16253994b7c4/attach?logs=1&stream=0&stdout=1 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/vnd.docker.raw-stream
    {{ STREAM }}
```

参数

- logs：其值为1/True/true或者0/False/false，表示是否返回日志，默认值为false。
- stream：其值为1/True/true或者0/False/false，表示是否返回数据流，默认值为false。
- stdin：其值为1/True/true或者0/False/false。如果stream=true，则附加标准输入stdin，默认值为false。
- stdout：其值为1/True/true或者0/False/false。若logs=1，返回标准输出，若stream=true，附加标准输出，默认值为false。
- stderr：其值为1/True/true或者0/False/false。若logs=true，返回标准错误日志，若stream=true，附加到标准错误输出，默认值为false。

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678/ attach?logs=1&stream=0 &stdout=1
```

13. 暂停容器

方法： POST /containers/(id)/pause

用例请求： POST /containers/e90e34656806/pause HTTP/1.1

用例返回： HTTP/1.1 204 No Content

参数：（无）

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/ e90e34656806/pause
```

14. 重新运行暂停容器

方法： POST /containers/(id)/unpause

用例请求： POST /containers/e90e34656806/unpause HTTP/1.1

用例返回：HTTP/1.1 204 No Content

参数：（无）

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/ e90e34656806/ unpause
```

15. 等待容器停止

方法：POST /containers/(id)/wait

用例请求：POST /containers/16253994b7c4/wait HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {"StatusCode":0}
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/16253994b7c4/wait
```

16. 删除容器

方法：DELETE /containers/(id)

用例请求：DELETE /containers/16253994b7c4?v=1 HTTP/1.1

用例返回：HTTP/1.1 204 No Content

参数

- v：其值为1/True/true或者0/False/false，用于删除关联的数据卷，默认值为false。
- force：其值为1/True/true或者0/False/false，用于强制删除，默认值为false。

状态码：（略）

curl操作：

```
curl -X DELETE http://localhost:2376/containers/4fa6e0f0c678
```

17. 从容器中复制目录或文件

方法：POST /containers/(id)/copy

用例请求：

```
POST /containers/4fa6e0f0c678/copy HTTP/1.1
    Content-Type: application/json
    {
         "Resource":"test.txt"
    }
```

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/x-tar
    {{ TAR STREAM }}
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/containers/4fa6e0f0c678//copy
```

B.2　镜像相关 API

镜像相关API主要和镜像的操作相对应，包括列出镜像、创建镜像、查看镜像详细信息等。

1. 列出镜像

方法：GET /images/json

用例请求：GET /images/json?all=0 HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    [
        {
            "RepoTags": [
                "ubuntu:12.04",
                "ubuntu:precise",
                "ubuntu:latest"
            ],
            "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "Created": 1365714795,
            "Size": 131506275,
            "VirtualSize": 131506275
        },
        ...
    ]
```

参数

- all：其值为1/True/true或者0/False/false，表示是否返回所有，默认值为false。

- filters：JSON形式的条件过滤。

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/ images/json?all=0
```

2. 创建镜像

方法：POST /images/create

用例请求：POST /images/create?fromImage=base HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {"status":"Pulling..."}
    {"status":"Pulling", "progress":"1 B/ 100 B", "progressDetail":{"current":1, "total":100}}
    {"error":"Invalid..."}
    ...
```

参数

- fromImage：基础镜像。
- fromSrc：导入源，一个可以获得镜像的URL地址。
- repo：仓库。
- tag：标签。
- registry：拉取镜像的注册服务器。

请求头

- X-Registry-Auth - base64-encoded：编码的认证对象。

状态码：（略）

curl操作：

```
curl -X POST http://localhost:2376/ images/create?fromImage=base
```

3. 查看镜像详细信息

方法：GET /images/(name)/json

用例请求：GET /images/base/json HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {
```

```
        "Created":"2013-03-23T22:24:18.818426-07:00",
        "Container":"3d67245a8d72ecf13f33dffac9f79dcdf70f75acb84d308770391510e0c23ad0",
        "ContainerConfig":
        {
            "Hostname":"",
            "User":"",
            "Memory":0,
            "MemorySwap":0,
            "AttachStdin":false,
            "AttachStdout":false,
            "AttachStderr":false,
            "PortSpecs":null,
            "Tty":true,
            "OpenStdin":true,
            "StdinOnce":false,
            "Env":null,
            "Cmd": ["/bin/bash"],
            "Dns":null,
            "Image":"base",
            "Volumes":null,
            "VolumesFrom":"",
            "WorkingDir":""
        },
        "Id":"b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
        "Parent":"27cf784147099545",
        "Size": 6824592
    }
```

参数：（无）

状态码

- 404：无该镜像。

curl操作：

```
curl -X GET http://localhost:2376/ images/base/json
```

4. 获得镜像历史

方法：GET /images/(name)/history

用例请求：GET /images/base/history HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    [
        {
            "Id":"b750fe79269d",
            "Created":1364102658,
            "CreatedBy":"/bin/bash"
        },
```

```
    {
        "Id":"27cf78414709",
        "Created":1364068391,
        "CreatedBy":""
    }
]
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376/ images/ base/history
```

5. 推送一个镜像到注册服务器

需要注意的是，如果你需要将一个镜像推送到一个私有注册服务器，那么该镜像必须是在引用该注册服务器域名和端口的库标记过，库名也必须加上URL路径。

方法： POST /images/(name)/push

用例请求： POST /images/test/push HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    {"status":"Pushing..."}
    {"status":"Pushing", "progress":"1/? (n/a)", "progressDetail":{"current":1}}}
    {"error":"Invalid..."}
    ...
```

参数：（无）

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376/ images/test/push
```

6. 给镜像贴标签并入库

方法： POST /images/(name)/tag

用例请求： POST /images/test/tag?repo=myrepo&force=0&tag=v42 HTTP/1.1

用例返回： HTTP/1.1 201 OK

参数

- repo：镜像签入的库。
- force：其值为1/True/true或者0/False/false，表示是否强制更改，其默认值为false。

- tag：新的标签名。

状态码

- 201：返回正常。
- 400：错误参数。
- 404：无该镜像。
- 409：冲突。
- 500：服务器错误。

curl操作：

```
curl -X POST http://localhost:2376 /images/test/tag?repo=myrepo&force=0&tag=v42
```

7. 删除镜像

方法：DELETE /images/(name)

用例请求：DELETE /images/test HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-type: application/json
    [
        {"Untagged":"3e2f21a89f"},
        {"Deleted":"3e2f21a89f"},
        {"Deleted":"53b4f83ac9"}
    ]
```

参数

- force：其值为1/True/true或者0/False/false，默认值为false。
- noprune：其值为1/True/true或者0/False/false，默认值为false。

状态码：（略）

curl操作：

```
curl -X DELETE http://localhost:2376 /images/test
```

8. 搜索镜像

方法：GET /images/search

用例请求：GET /images/search?term=sshd HTTP/1.1

用例返回：

```
HTTP/1.1 200 OK
    Content-Type: application/json
    [
```

```
        {
            "description": "",
            "is_official": false,
            "is_automated": false,
            "name": "wma55/u1210sshd",
            "star_count": 0
        },
        {
            "description": "",
            "is_official": false,
            "is_automated": false,
            "name": "jdswinbank/sshd",
            "star_count": 0
        },
        {
            "description": "",
            "is_official": false,
            "is_automated": false,
            "name": "vgauthier/sshd",
            "star_count": 0
        }
        ...
    ]
```

参数

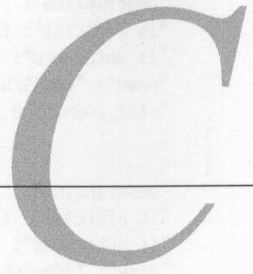
- term：搜索关键字。

状态码：（略）

curl操作：

```
curl -X GET http://localhost:2376 //images/search?term=sshd
```

附录 C 参考资料



1. http://blog.flux7.com/blogs/docker/docker-tutorial-series-part-7-ultimate-guide-for-docker-apis
2. https://clusterhq.com/blog/fig-flocker-multi-server-docker-apps/
3. https://docs.clusterhq.com/en/latest/gettingstarted/installation.html
4. http://www.vpsee.com/2013/11/shipyard-a-docker-web-ui/
5. http://vmking.blog.51cto.com/6433018/1537330
6. http://serverascode.com/2014/05/25/docker-shipyard-multihost.html
7. http://shipyard-project.com/docs/quickstart/
8. http://devopscube.com/docker-tutorial-getting-started-with-docker-swarm/
9. http://blog.daocloud.io/swarm_analysis_part1/
10. https://github.com/llitfkitfk/docker-tutorial-cn/blob/master/intro-to-docker-swarm-pt2-config-options-requirements-cn.md
11. http://technolo-g.com/intro-to-docker-swarm-pt4-demo/
12. http://blog.remmelt.com/2014/12/07/docker-swarm-setup/
13. http://devopscube.com/docker-tutorial-getting-started-with-docker-swarm/
14. https://github.com/docker/docker/issues/9459
15. https://www.docker.com/
16. http://blog.daocloud.io/how-to-master-docker-image/
18. http://soft.zdnet.com.cn/techupdate/2008/0317/772069.shtml
19. http://docs.docker.com/articles/https/
20. http://www.21ops.com/linux/13512.html
21. http://segmentfault.com/a/1190000000801162

22. http://blog.csdn.net/wsscy2004/article/details/26279569

23. http://market.aliyun.com/products/55528001/jxsc000057.html

24. http://help.aliyun.com/knowledge_detail.htm?knowledgeId=5974865

25. https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

26. http://www.expressjs.com.cn/4x/api.html#application

27. http://www.infoq.com/cn/news/2014/09/docker-safe

28. http://blog.csdn.net/wangpengqi/article/details/9821227

29. http://laokaddk.blog.51cto.com/368606/674256

Docker
开发实践

Docker是当之无愧的Go语言杀手级应用，并且现如今Docker这个词的含义越来越丰富了，以至于它已经代表了容器技术的生态圈。本书奉行实践出真知，其中的案例都非常棒。更为关键的是，其中的高级篇对Docker生态圈的各个新成员也做了非常翔实的介绍和实践，这真的很难能可贵。

——郝林，Go语言北京用户组发起人，《Go并发编程实战》作者

腾讯的互娱的开发节奏，只有Docker跟得上！如果你想你的团队加快开发速度，那么我推荐你使用Docker，而本书从基础、案例到高级话题，都有很全面的覆盖。

——易剑，腾讯互动娱乐事业群高级架构师

这是一本关于Docker的好书，值得所有想了解Docker的人放在键盘左边。

——李毅秋，人人网技术总监

云计算的初级是数据的云化，下一步是程序的云化，而Docker则是当前程序云化最好的工具。让你的程序一次配置，全网增量迁移、运行。本书出自一线互联网研发人员之手，它是实战的结晶，所涉案例都是互联网公司的真实应用，对Docker的应用都不是浅尝辄止，而是带你登堂入室。

——潘向荣，迅雷看看高级技术经理

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/云计算/Docker
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39519-1
定价：59.00元